Юрий Магда

bhv

UNIX

- Базовые концепции UNIX
- Работа в сетях TCP/IP
- Электронная почта и Интернет
- Графические оболочки UNIX
- Редакторы текста
- Командный интерпретатор Shell
- Разработка приложений на C и Perl

Наиболее полное руководство

В ПОДЛИННИКЕ®

Юрий Магда

UNIX

Санкт-Петербург

«БХВ-Петербург»

2006

УДК 681.3.066
ББК 32.973.26-018.2
М12

Магда Ю. С.

М12 UNIX. — СПб.: БХВ-Петербург, 2006. — 528 с.: ил.

ISBN 5-94157-824-5

Рассматривается широкий круг вопросов функционирования операционной системы UNIX. Анализируются принципы взаимодействия процессов, управления учетными записями пользователей и построения файловой системы. Изложены базовые концепции функционирования и настройки сетей на основе протокола TCP/IP и их реализация в операционных системах UNIX. С позиции пользователя описаны современные методы обработки текстовой документации и работа с графическими оболочками операционной системы. Значительная часть материала книги посвящена основам разработки приложений на языках C и Perl в среде UNIX, созданию командных файлов в интерпретаторе shell.

Теоретические аспекты функционирования UNIX иллюстрируются многочисленными примерами программ, разработанных на языке C.

Для пользователей UNIX

УДК 681.3.066
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 15.06.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 42,57.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-824-5

Оглавление

Благодарности

Автор выражает огромную благодарность сотрудникам издательства "БХВ-Петербург" за подготовку материалов книги к изданию. Особая признательность жене Юлии за неоценимую помощь и поддержку при подготовке рукописи.

Введение

В данной книге рассматриваются основные принципы функционирования операционной системы UNIX, а также практические аспекты настройки рабочей среды пользователя и основы разработки программного обеспечения. По этой теме издано достаточно много книг, предназначенных для очень широкой аудитории, начиная новичками и заканчивая опытными пользователями и системными администраторами.

Отличие этой книги от других подобных изданий состоит в том, что в ней рассматривается очень широкий круг вопросов, касающихся как функционирования самой операционной системы UNIX, так и работы пользователя в ней. Большинство книг по UNIX имеет, как правило, конкретную направленность и предназначено для определенного круга читателей. Так, в литературе для новичков основной акцент делается, как правило, на изучении пользовательского интерфейса. Системных администраторов и разработчиков программного обеспечения больше интересуют принципы построения и функционирования операционной системы, которые описаны в книгах для профессиональных пользователей.

В этой книге делается попытка объединить различные подходы к анализу операционной системы, рассматривая программную архитектуру UNIX, программный интерфейс разработчика и интерфейс пользователя как части единого целого, а именно операционной системы UNIX. По этой причине многие аспекты взаимодействия пользователя и операционной системы иллюстрируются примерами программного кода на языке C, что, по мнению автора, способствует глубокому пониманию функционирования различных подсистем UNIX и осмысленным действиям по конфигурированию и настройке операционной системы пользователем. Тем не менее, от читателя не требуется глубоких знаний по программированию на C, достаточно знать основы этого языка.

Материал книги охватывает следующий круг вопросов:

- анализ теоретических основ функционирования операционных систем UNIX, позволяющий понять те или иные практические приемы работы, а также смысл параметров, используемых для настройки;
- настройка и использование графического интерфейса пользователя и программ обработки текстовой документации;
- конфигурирование и настройка сети в операционной системе UNIX, а также теоретические и практические аспекты функционирования Интернета и электронной почты;
- принципы разработки программного обеспечения в UNIX с использованием языков C и Perl.

Книга состоит из двенадцати глав, краткое описание каждой из них приведено далее.

- *Глава 1. "Обзор операционных систем UNIX".*

  В главе рассматриваются стандарты операционных систем UNIX и проводится сравнительный анализ наиболее популярных программных продуктов, таких как Linux, FreeBSD и Solaris. Рассмотрены также и наиболее распространенные аппаратные платформы, на которых работает UNIX.

- *Глава 2. "Архитектура UNIX".*

  Материал главы посвящен обзору архитектуры операционных систем UNIX. Рассматриваются принципы построения UNIX, взаимодействие ядра с другими подсистемами, а также механизмы запуска и выполнения программ пользователя. Здесь же анализируются механизмы реализации многозадачности и многопользовательского режима. Подробно рассматриваются механизмы взаимодействия процессов, выполняющихся в операционной системе, с ядром.

- *Глава 3. "Учетные записи пользователей".*

  Здесь рассмотрены вопросы управления учетными записями пользователей: регистрация, изменение и удаление учетных записей пользователей, а также получение информации о пользователях.

- *Глава 4. "Командный интерпретатор shell".*

  В этой главе рассматривается использование встроенного командного интерпретатора shell. Приводится описание синтаксиса shell, включая описание переменных и логических программных структур интерпретатора. Представлены многочисленные примеры применения командного интерпретатора shell для решения различных задач.

- *Глава 5. "Файловая система UNIX".*

  Эта глава посвящена администрированию файловой системы. Рассмотрены организация и структура файловых систем, операции с файлами и каталогами, а также назначение и изменение прав доступа к объектам файловой системы. Анализируются практические аспекты создания и управления файловыми системами, монтирование и демонтирование файловых систем.

- *Глава 6. "Обработка текста в UNIX".*

  Материал главы знакомит читателя с популярными программами для обработки текстовой документации, раскрывает широкий диапазон возможностей современных текстовых редакторов. Подробное описание различных режимов работы этих приложений значительно облегчает усвоение принципов функционирования и настройки текстовых редакторов.

- *Глава 7. "Процессы в UNIX".*

  В главе рассматривается один из ключевых аспектов функционирования операционной системы — взаимодействие процессов UNIX и управление ими. Здесь рассмотрены концепции создания, жизненного цикла и уничтожения процессов, а также программные средства для мониторинга процессов и получения статистической информации.

- *Глава 8. "Поддержка сетей в UNIX".*

  Глава знакомит читателя с принципами функционирования операционной системы UNIX в сетях TCP/IP. Детально рассматриваются основы построения сетей и основные протоколы взаимодействия сетевых приложений, а также вопросы настройки и конфигурирования протокола TCP/IP в операционной системе UNIX. Проанализированы принципы функционирования сетевых сервисов операционной системы UNIX, таких как DNS, DHCP и NFS. Значительная часть материала посвящена тестированию сетей и поиску неисправностей.

- *Глава 9. "Электронная почта".*

  Материал главы посвящен анализу работы и настройке систем электронной почты в UNIX. Здесь рассматриваются теоретические основы построения таких систем, а также анализируются основные аспекты использования наиболее популярных программ электронной почты. В частности, даются рекомендации по настройке базовой программы для пересылки электронной почты `sendmail`. В главе описывается функционирование основных протоколов электронной почты — SMTP, POP3 и IMAP4 и тестирование их работоспособности в UNIX-системах.

- *Глава 10. "UNIX и Интернет".*

  Эта глава посвящена вопросам работы систем UNIX в Интернете. Здесь описан протокол HTTP, являющийся базисным при обмене информацией между клиентами и Web-узлами, а также приводится пример настройки и тестирования популярного Web-сервера Apache. Часть материала главы посвящена примерам программирования простейших интернет-приложений.

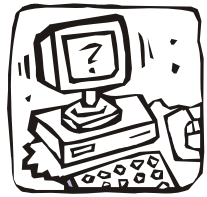
- *Глава 11. "Графические оболочки UNIX".*

  Здесь рассматриваются основы построения графических интерфейсов UNIX-систем. Значительная часть материала главы посвящена конфигурированию и настройке графической системы X Window, являющейся основой создания графических приложений для операционных систем UNIX. Кроме этого, рассматриваются основные принципы функционирования и настройки популярных графических оболочек GNOME и KDE.

- *Глава 12. "Разработка приложений в среде UNIX".*

  В главе обсуждаются наиболее важные аспекты разработки программного обеспечения для систем UNIX с использованием наиболее популярных языков программирования C и Perl. Здесь рассмотрены принципы компиляции и сборки приложений с использованием популярного компилятора g++. Ввиду большой популярности языка Perl значительная часть материала главы посвящена описанию синтаксиса языка с объяснением примеров программ.

Материал книги окажет существенную помощь читателям, имеющим определенный опыт работы с операционными системами UNIX, которые хотели бы существенно пополнить свои знания в этой области. Книга может быть полезна также системным администраторам и разработчикам программного обеспечения.



Глава 1

Обзор операционных систем UNIX

С момента своего появления операционная система UNIX стала довольно популярной и получила широкое распространение в вычислительных системах различной производительности, начиная от персональных компьютеров и заканчивая большими вычислительными комплексами. Такая популярность этой операционной системы обусловлена несколькими факторами.

Во-первых, это более чем трехдесятилетний цикл развития. За этот период операционная система UNIX выдержала проверку временем и проявила себя как очень эффективная программная среда. Во-вторых, программный код системы полностью написан на языке высокого уровня C, что делает ее понятной для пользователей и разработчиков программного обеспечения. Это позволяет относительно легко вносить изменения как в существующие версии UNIX, так и переносить операционную систему на другие аппаратные платформы. Кроме этого, во многих случаях версии этой операционной системы поставляются вместе с исходными текстами, что позволяет легко адаптировать UNIX под специфические требования, после чего перекомпилировать систему.

Изначально операционная система UNIX создавалась как многопользовательская и многозадачная система, ориентированная в первую очередь на выполнение серверных или управляющих функций для клиентских компьютеров. Такие подходы, а также мощные встроенные средства удаленного администрирования, способствовали тому, что UNIX заняла лидирующие позиции на рынке интернет-серверов и серверов баз данных. Немаловажную роль в популяризации операционной системы сыграла и структура ее файловой системы, представляющая единую иерархическую систему с унифицированным доступом как к файлам данных, так и к аппаратным ресурсам, таким как диски, терминалы, принтеры, сеть, память и т. п.

В практическом плане операционная система UNIX предоставляет пользователю целый ряд преимуществ. Перечислим только некоторые из них:

- все популярные приложения (пакеты офисных программ, базы данных и др.) известных производителей программного обеспечения, как правило, реализованы для работы в операционной системе UNIX;
- UNIX поддерживает широкий спектр средств передачи информации, включая многочисленные сетевые и коммуникационные протоколы, что обеспечивает эффективную работу операционной системы в сетях;
- операционная система UNIX имеет весьма развитые средства системного и сетевого управления, в том числе эффективные инструменты для удаленного администрирования и настройки;
- UNIX является мощной платформой для разработки приложений, в нее включены наиболее популярные языки разработки приложений, средства работы с базами данных, а также другое программное обеспечение;
- операционная система поддерживает все основные аппаратные процессорные архитектуры, включая надежную поддержку SMP, MMP и кластерных систем. В других серверных средах такая поддержка отсутствует;
- практически все важнейшие промышленные, международные, официально утвержденные и неофициальные стандарты были впервые разработаны для UNIX и только впоследствии распространились и на другие операционные системы. В настоящее время основным стандартом является разработанная консорциумом X/Open Единая спецификация UNIX, содержащая более тысячи интерфейсов прикладных программ и поддерживаемая всеми основными производителями операционной системы UNIX. Несмотря на то, что различные версии UNIX обладают уникальными возможностями, все они отвечают требованиям стандарта POSIX (Portable Operating System Interface, переносимый интерфейс операционной системы), а также стандарту X/Open Portability Guide, Edition 4 (XPG 4) и сертифицированы X/Open на соответствие стандартам UNIX 93;
- операционная система UNIX к настоящему времени утвердилась как платформа для персональных приложений, приложений для рабочих групп и приложений корпоративного класса.

В настоящее время единого стандарта для UNIX не существует, и на рынке присутствует множество версий этой операционной системы, каждая из которых имеет свои названия и особенности. Тем не менее, все без исключения UNIX-системы имеют однотипную архитектуру, интерфейсы и среду программирования, что дает основание считать все UNIX-системы в той или иной степени родственными.

Простота и способность операционных систем UNIX к расширению и модификациям являются весьма серьезными преимуществами по сравнению с другими системами, поэтому UNIX стали переносить на множество платформ. Тем не менее, несмотря на множество реализаций базовой системы, среди них выделяются две основные ветви, берущие начало от System V UNIX и BSD UNIX.

Одна ветвь происходит от версий 4.2, 4.3 или 4.4BSD, другая базируется на системах SVR3 (System V Release 3) или SVR4 (System V Release 4). На протяжении ряда лет версия BSD пользовалась бóльшей популярностью в академических и научных кругах, в то время как версии System V, разработанные компанией AT&T, занимали лидирующие позиции в коммерческих организациях и в промышленности. Несмотря на существующие различия между этими основными типами UNIX-систем, подавляющее большинство пользовательских команд работает одинаково и имеет один и тот же синтаксис во всех версиях операционных систем, независимо от того, используется ли AIX, BSD, HP/UX, Linux или Solaris.

Существующие в настоящее время отличия между операционными системами не имеют принципиального значения, и определить, к какой из ветвей принадлежит та или иная реализация операционной системы, иногда бывает довольно сложно. Вот основные различия между операционными системами System V и BSD:

- разные способы установки и настройки терминальных устройств;
- разные способы инициализации, именования конфигурационных файлов и файлов инициализации системы;
- различная настройка параметров файловой системы;
- различные методы получения диагностической информации и ее отображения на консоли

и т. п.

С точки зрения пользователя принципиальных различий между разными ветками операционной системы UNIX не существует. Справедливости ради следует отметить, что BSD и System V — далеко не единственные реализации операционной системы UNIX. Рассмотрим особенности реализации некоторых, наиболее популярных версий операционной системы UNIX и начнем с Solaris.

Исходный код UNIX System V Release 4 был доработан компанией Sun, в результате чего появилась реализация операционной системы UNIX, названная Solaris. Эта операционная система имеет несколько основных отличий от ба-

зовой операционной системы. В частности, в Solaris были добавлены следующие возможности:

- многонитевость;
- симметричная многопроцессорная обработка;
- режим реального времени.

В настоящее время Solaris является одной из самых распространенных версий операционной системы UNIX и работает на платформах SPARC и Intel86. Для развития продукта фирма Sun Microsystems предоставила более открытый доступ к кодам операционной среды Solaris.

Другая ветвь UNIX — FreeBSD — берет свое название от "Berkeley Software Distribution". Эта операционная система основана на версии 4.4BSD-Lite и сохраняет специфичные черты модели развития BSD-систем.

На базе версии 4.4BSD-Lite было создано несколько операционных систем с открытыми исходными кодами, среди которых особо следует выделить проект GNU. Операционная система FreeBSD позволяет:

- выполнять одновременно несколько задач с динамическим регулированием приоритетов, что распределяет ресурсы компьютера между приложениями и пользователями оптимальным образом;
- одновременно работать многим пользователям и использовать систему совместно для решения ряда задач. Это означает, что системные ресурсы, такие, например, как принтеры и накопители на магнитных лентах, могут распределяться между пользователями в системе или сети, при этом для каждого пользователя или группы пользователей могут быть установлены определенные ограничения на использование того или иного ресурса. Это позволяет избежать перегрузок в работе операционной системы;
- работать с распространенными сетевыми протоколами и стандартами, такими как SLIP, PPP, NFS, DHCP и NIS. Это позволяет операционной системе FreeBSD эффективно функционировать совместно с другими операционными системами, например, Windows. Кроме того, FreeBSD может использоваться в качестве интернет-сервера, предоставляя полный спектр сервисов (WWW, FTP, маршрутизация);
- использовать стандарт X Window System (X11R6), предоставляющий графический интерфейс пользователю.

Операционная система FreeBSD обеспечивает совместимость на уровне программного кода с большинством программ, разработанных для Linux и System V Release 4, обладая полным комплектом инструментальных средств для разработки программ (языки C, C++, Fortran и Perl). Исходные

тексты FreeBSD свободно распространяются через Интернет, так что систему можно оптимизировать для специальных приложений или проектов. Для коммерческих операционных систем такая возможность отсутствует.

FreeBSD очень часто используется как платформа для высокопроизводительных рабочих станций, при этом она оказывается более эффективной по сравнению с другой, не менее популярной операционной системой Linux. Системы на основе BSD могут демонстрировать бóльшую по сравнению с Linux производительность, обеспечивая при этом более высокую надежность. Наконец, операционная система FreeBSD может выполнять код, разработанный для Linux, но не наоборот.

Еще одной, очень популярной реализацией UNIX является Linux. Эта версия UNIX обладает большинством свойств, присущих другим реализациям, и, кроме того, включает некоторые дополнительные возможности. Linux — это полная многозадачная многопользовательская операционная система, допускающая одновременную работу многих пользователей.

Операционная система Linux очень популярна среди миллионов пользователей. GNU/Linux вместе с набором инструментальных средств по оценкам экспертов охватывает около 40% рынка UNIX. Многие компании выпускают дистрибутивы Linux — пакеты, включающие ядро, множество утилит, приложений и программное обеспечение для установки ОС. GNU/Linux получила поддержку у таких компаний, как Sun и IBM.

Для разработки программного обеспечения, работающего в Linux, был создан специальный фонд под названием Free Software Foundation (FSF), цель которого заключается в поиске источников финансирования разработки программного обеспечения GNU. Несмотря на относительно короткую историю существования, под эгидой проекта GNU было создано и адаптировано огромное количество программ, среди которых наиболее известными являются утилиты Emacs, gcc (компилятор GNU C) и bash (командная оболочка).

Необходимо отметить, что эта операционная система хорошо совместима с рядом стандартов для UNIX на уровне исходных текстов, включая IEEE POSIX.1, System V и BSD, поскольку создавалась с расчетом на такую совместимость.

Большинство из свободно распространяемого через Интернет программного обеспечения для UNIX может быть откомпилировано для работы в Linux с минимальными изменениями. Более того, все исходные тексты для Linux, включая ядро, драйверы устройств, библиотеки, пользовательские программы и инструментальные средства, также распространяются свободно.

К специфическим особенностям Linux следует отнести контроль работ по стандарту POSIX (используемый оболочками, такими как csh и bash), работу

с псевдотерминалами, поддержку национальных и стандартных клавиатур с динамически загружаемыми драйверами клавиатур.

Операционная система Linux поддерживает различные типы файловых систем. Некоторые из них, такие, например, как файловая система ext2fs, были созданы специально для Linux. Кроме того, поддерживаются и другие типы файловых систем, такие, например, как Minix и Xenix. Система включает реализацию файловой системы MS-DOS, позволяющую прямо обращаться к файлам MS-DOS на жестком диске. Наконец, для работы с CD-ROM поддерживается стандарт файловой системы ISO 9660.

Как и другие операционные системы, Linux обеспечивает полный набор протоколов TCP/IP для сетевой работы. Сюда входят драйверы устройств для многих сетевых карт Ethernet, SLIP (Serial Line Internet Protocol, обеспечивающий пользователям доступ по TCP/IP при последовательном соединении), PLIP (Parallel Line Internet Protocol), PPP (Point-to-Point Protocol), NFS и т. д. В систему включена поддержка всего спектра сервисов TCP/IP (FTP, telnet, NNTP и SMTP).

Глава 2

Архитектура UNIX

В этой главе мы рассмотрим базовые концепции построения операционных систем UNIX и взаимодействие различных функциональных частей. Знание основных принципов функционирования системы является основой для успешной работы в UNIX и помогает эффективно использовать возможности этой мощной операционной системы. Кроме того, многие принципы построения UNIX используются в других популярных операционных системах, например, Windows, что может помочь и при изучении таких систем.

В основу построения операционных систем UNIX было положено несколько важных принципов.

- **Надежность.** Операционная система должна быть по возможности максимально надежной. Сбои в работе системы в большинстве случаев могут вызываться неполадками в аппаратной части, неправильными действиями пользователя либо некорректной работой программного обеспечения. Во всех этих случаях для повышения надежности требуется каким-то образом сохранять работоспособность системы, обеспечивая определенный минимальный уровень функционирования и возможности восстановления. Самым неприятным следствием сбоя в работе может быть потеря важных пользовательских данных, поэтому сохранение и восстановление данных является основным критерием надежности системы. Одним из эффективных методов повышения надежности системы является разделение программного кода операционной системы и кода пользовательской программы таким образом, чтобы программа пользователя не могла разрушить выполняющийся программный код UNIX. С другой стороны, пользовательская программа должна иметь доступ к различным ресурсам операционной системы: памяти, процессору, жестким дискам, периферийным устройствам (принтерам, плоттерам, накопителям на магнитных лентах и т. д.), что является потенциально опасным для надежного функционирования системы.

Компромиссным решением стала концепция построения операционной системы на основе базового программного модуля ("ядра"), выполняющего обслуживание запросов программ пользователя, одновременно изолируя их от прямого доступа к ресурсам системы.

- **Возможность одновременной работы нескольких пользователей с одной системой.** В этом случае необходимо предоставлять ресурсы операционной системы (возможно, одни и те же) нескольким пользователям одновременно, причем возможности доступа к одним и тем же ресурсам для разных пользователей могут отличаться. Операционные системы, допускающие работу в таком режиме, называются многопользовательскими. Очень часто разным пользователям требуется доступ к одним и тем же ресурсам, например, к файлу на диске. При этом одни пользователи могут читать и записывать данные в файл, в то время как другие могут только читать данные из файла или вообще не иметь доступа к данным. Многопользовательская операционная система должна обладать механизмами разделения доступа к ресурсам и, кроме того, иметь возможности для защиты общих ресурсов от несанкционированного доступа. UNIX относится именно к такому классу операционных систем.
- **Возможность одновременного выполнения множества процессов.** В операционной системе должна быть возможность выполнения одновременно множества процессов (работающих программ), как пользовательских, так и системных. При этом должен обеспечиваться механизм синхронизации процессов. Это означает, что операционная система должна контролировать запуск, выполнение и уничтожение процессов, обеспечивая работающим процессам доступ к требуемым ресурсам наиболее эффективным способом (мультизадачность).
- **Унифицированный (единообразный) способ доступа к ресурсам операционной системы.** Эта концепция положена в основу построения файловой системы UNIX. В операционной системе UNIX объектами файловой системы являются как файлы программ или данных, так и устройства, например, принтеры, жесткие диски, терминальные линии. Подобный подход очень удобен, поскольку позволяет работать с единым интерфейсом и использовать одни и те же функции как для работы с файлами данных, так и с файлами устройств. Существенным преимуществом такого подхода является и то, что можно использовать единые принципы для разделения ресурсов системы в многопользовательской среде и установки защиты объектов файловой системы. Кроме того, разработчики программного обеспечения могут использовать единый интерфейс для разработки программ, что значительно снижает трудоемкость.

Реализованная на основе этих концепций операционная система UNIX обладает следующими характеристиками:

- ❒ она легко переносима на другие аппаратные платформы;
- ❒ допускает работу в режиме вытесняющей многозадачности, обеспечивая работу процессов в изолированных адресных пространствах в виртуальной памяти;
- ❒ обеспечивает поддержку одновременной работы многих пользователей;
- ❒ поддерживает работу асинхронных процессов;
- ❒ имеет иерархическую файловую систему;
- ❒ обеспечивает поддержку независимых от устройств операций ввода/вывода путем использования специальных файлов устройств;
- ❒ предоставляет стандартный интерфейс для программ (программные каналы) и пользователей (командный интерпретатор, не входящий в ядро операционной системы);
- ❒ имеет встроенные средства мониторинга операционной системы.

Посмотрим, как отображены вышеперечисленные возможности операционной системы UNIX в ее программной архитектуре. В наиболее общем виде операционную систему UNIX можно представить пятиуровневой моделью (рис. 2.1).

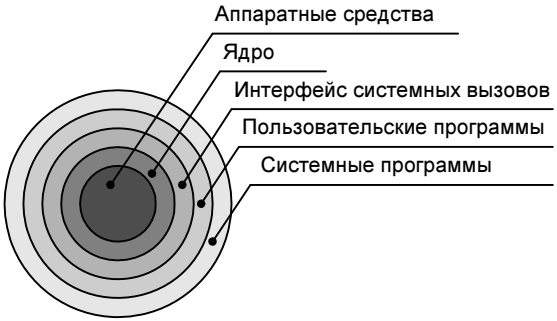


Рис. 2.1. Архитектура UNIX

Как видно из рис. 2.1, операционную систему UNIX можно представить как совокупность пяти взаимосвязанных функциональных частей:

- ❒ аппаратной части (hardware);
- ❒ ядра;
- ❒ интерфейса системных вызовов;

- системных служебных программ (утилит) и командных оболочек (командных интерпретаторов);
- пользовательских программ.

Аппаратная часть представляет собой физические ресурсы системы (процессор, оперативная память, жесткий диск, устройства ввода/вывода), непосредственный доступ к которым может осуществлять только ядро операционной системы. Прикладные пользовательские программы не могут получить прямой доступ к оборудованию системы, а взаимодействуют с ним посредством ядра, вернее через системные вызовы функций ядра.

Такое построение операционной системы обусловлено несколькими причинами. Во-первых, это гарантирует надежность работы системы, поскольку невозможно произвольным образом, например, из программы пользователя, изменить конфигурацию системы, что чревато крахом UNIX. Во-вторых, ядро операционной системы балансирует работу всех подсистем без вмешательства пользователя, обеспечивая тем самым оптимальную производительность работы. В-третьих, ограничение доступа к аппаратным средствам пользовательских программ обеспечивает надежную работу аппаратуры, поскольку ядро управляет функционированием физических устройств посредством специальных программ-драйверов устройств.

Операционная система UNIX является многопользовательской и многозадачной системой. Это означает, что в ней могут работать одновременно несколько пользователей, каждый из которых может выполнять несколько задач одновременно.

В основе функционирования операционной системы UNIX, как было упомянуто, лежит взаимодействие между пользовательскими программами, ядром и аппаратными ресурсами. Фактически функционирование операционной системы определяется особенностями работы ядра, поэтому остановимся на взаимодействии ядра и остальных функциональных частей UNIX более подробно.

Компиляция и сборка ядра операционной системы UNIX обычно выполняются статически. Это означает, что ядро загружается как один большой исполняемый программный модуль при инициализации системы. Такой тип ядра называют монолитным. Некоторые версии операционных систем допускают работу с другим типом ядра, который называют модульным или, иногда, микроядром. При использовании модульного ядра дополнительные модули программного кода (обычно это драйверы устройств) подгружаются в оперативную память динамически, т. е. по мере необходимости, например, при включении аппаратного устройства. Такие дополнительные модули реализованы в виде загружаемых модулей ядра.

Преимущество модульного ядра состоит в том, что базовый модуль ядра имеет небольшой размер, быстрее загружается и требует меньше ресурсов операционной системы. В то же время монолитное ядро работает чуть быстрее, поскольку не требуется переключений контекста выполняемых процессов (что имеет место в случае загрузки/выгрузки дополнительных модулей) и дополнительной синхронизации, как при использовании отдельных модулей. Кроме того, в монолитном ядре реализовано намного больше функций управления аппаратурой, поскольку такое ядро содержит драйверы устройств.

В настоящее время большинство ядер UNIX реализованы как комбинированные, т. е., являясь в принципе монолитными, допускают загрузку дополнительных модулей во время работы операционной системы.

Современные версии ядра UNIX-систем, в большинстве своем, обладают еще одной особенностью. Эта особенность — возможность функционирования ядра как совокупности отдельно выполняющихся потоков (kernel threads). Подобная особенность называется многопоточностью ядра и позволяет повысить эффективность функционирования как ядра, так и всей операционной системы. В первом приближении поток можно представить себе как отдельный выполняющийся фрагмент программного кода в рамках одного процесса. При этом ядро управляет выполнением потоков и их синхронизацией.

Более высокая эффективность выполнения многопоточных функций обусловлена тем, что переключение контекста отдельных потоков требует меньше времени, чем переключение контекста отдельных процессов. В значительной степени подобный выигрыш получается за счет того, что потоки выполняются в общем адресном пространстве, в то время как каждый процесс требует отдельного адресного пространства, а это занимает определенное время.

Особенностью современных операционных систем UNIX является еще и то, что ядро таких систем, кроме обработки отдельных потоков, поддерживает также работу многопоточных пользовательских программ, что повышает их производительность. В этом случае многопоточные приложения выполняются как совокупность элементарных (lightweight) процессов, которые используют общее адресное пространство, общие страницы памяти и открытые файлы. Естественно, что для получения выигрыша в производительности выполняющаяся программа должна поддерживать реализацию многопоточности.

Большинство реализаций ядра операционной системы UNIX выполняется в режиме невытесняющей многозадачности (non-preemptive multitasking). Это означает, что операционная система не может прерывать выполнение процесса, выполняющегося в режиме ядра. Ядро, работающее в режиме вытес-

няющей многозадачности (preemptive multitasking), используется, как правило, в UNIX-подобных операционных системах, функционирующих в режиме реального времени. Некоторые популярные операционные системы, например, Sun 2.*x*, используют такой тип ядра.

Независимо от архитектуры ядро UNIX-системы обеспечивает поддержку многопользовательского и многозадачного режима работы, выполняя следующие функции:

- создание, выполнение, остановку и завершение процессов, а также синхронизацию их взаимодействия;
- планирование приоритетов выполнения процессов путем выделения им времени центрального процессора. В этом случае центральный процессор выполняет процесс в течение определенного ядром интервала времени, после чего процесс приостанавливается, и ядро начинает выполнение другого процесса. Через определенный интервал времени приостановленный процесс возобновляет выполнение и т. д.;
- выделение исполняемому процессу определенного объема оперативной памяти. В этом случае ядро защищает адресное пространство процесса от доступа из других процессов, одновременно позволяя разным процессам совместно использовать участки адресного пространства на определенных условиях. Если системе требуется некоторый объем свободной памяти, ядро освобождает память за счет процесса. При этом контекст (данные и ссылки) процесса сохраняется на жестких дисках или иных внешних устройствах. Такая реализация системы UNIX называется системой со свопингом (подкачкой). Если же на жестком диске сохраняются страницы памяти, то такая система называется системой с замещением страниц;
- выделение памяти на устройствах постоянного хранения информации (жесткие диски и магнитные ленты) для обеспечения эффективного хранения и выборки данных пользователя. Эта функция ядра реализуется через обращение к файловой системе UNIX. Файловая система управляется посредством функций ядра, которые выделяют внешнюю память для файлов, выполняют отображение физической структуры файловой системы в логическую форму, доступную для работы пользователей, а также устанавливают атрибуты доступа к объектам файловой системы, защищая пользовательские файлы от несанкционированного доступа;
- управление доступом процессов к периферийным устройствам, таким как терминалы, накопители на магнитных лентах и сетевое оборудование.

С точки зрения пользователя функции ядра можно представить так, как показано на рис. 2.2.

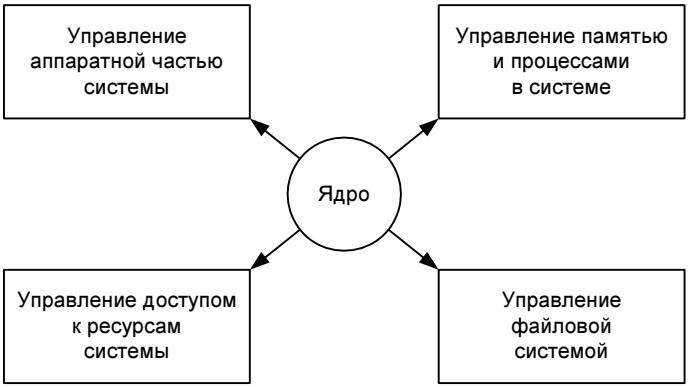


Рис. 2.2. Функции ядра UNIX

Ядро операционной системы является прозрачным для пользовательской программы. Это означает, что детали взаимодействия программы пользователя и операционной системы скрыты от пользователя. К примеру, если программа пользователя обращается к какому-либо файлу и записывает в него данные, то ядро системы выполняет последовательность довольно сложных действий: определяет местоположение файла на носителе, получает информацию о расположении требуемых данных в физических секторах накопителя, определяет место записи блока данных в физическую область дискового пространства и т. д. Наконец, ядро вызывает драйвер устройства и передает ему параметры и код операции (записи данных), после чего и выполняется запись данных.

Кроме вышеперечисленных ядро реализует ряд необходимых функций по обеспечению выполнения процессов пользовательского уровня, за исключением функций, которые реализуются на самом пользовательском уровне.

Например, ядро выполняет определенные действия, необходимые для работы командного интерпретатора shell. Такие функции командного интерпретатора, как чтение вводимых с терминала данных, динамическое создание процессов, синхронизация выполнения процессов, открытие программных конвейеров и переадресация ввода/вывода — все они реализуются через системные вызовы ядра.

Здесь я хочу сделать небольшое отступление и чуть более подробно остановиться на некоторых терминах и понятиях, часто используемых при анализе функционирования операционной системы UNIX. Эти термины будут встречаться очень часто и являются фундаментальными при анализе операционной системы. К таким терминам относятся "программа" и "процесс".

Под термином "программа" мы будем понимать записанный на носителе исполняемый файл, в то время как термин "процесс" в первом приближении будет означать программу, находящуюся в стадии выполнения. Ввиду важности понятия "процесс" остановимся на нем подробнее.

Процесс можно представить как исполняемый модуль программного кода, которому предоставлены определенные ресурсы системы (память, процессорное время и т. д.). В операционной системе UNIX может одновременно выполняться множество процессов (многозадачность), причем их число логически не ограничивается, и одна программа может создавать множество процессов. При этом существующие в системе процессы могут создавать новые или завершать другие процессы. Ядро операционной системы синхронизирует выполнение этапов процесса и управляет реакцией на наступление различных событий. Благодаря наличию системы защиты процессы выполняются независимо и не влияют друг на друга.

Создание и выполнение процессов иллюстрирует рис. 2.3.

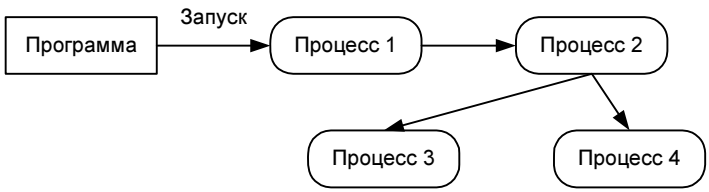


Рис. 2.3. Функционирование процессов в операционной системе UNIX

В простейшем случае программа порождает один процесс, в других случаях таких процессов может быть множество. Термин "процесс" применим не только к пользовательским или системным программам, но и к ядру, поскольку оно само функционирует как совокупность взаимосвязанных процессов.

Все процессы, выполняющиеся в среде операционной системы UNIX, могут выполняться либо в пользовательском режиме (User Mode), либо в режиме ядра (Kernel Mode). Подобное разделение обусловлено архитектурой системы и возможностью доступа процессов к ресурсам системы *(об этом упоминалось ранее в этой главе)*. Пользовательский режим не позволяет напрямую обращаться к аппаратным ресурсам системы и системным структурам данных, в то время как процесс, выполняющийся в режиме ядра, имеет такую возможность.

В пользовательском режиме могут работать не только программы пользователя, но и значительная часть системных программ, входящих в состав операционной системы.

Возникает вопрос: каким образом пользовательский процесс в случае необходимости может получить доступ к ресурсам системы?

Операционная система UNIX предоставляет процессам, работающим в режиме пользователя (User Mode), набор интерфейсов для взаимодействия с аппаратными устройствами, такими как процессор, жесткие диски, принтеры и т. д. UNIX реализует такие интерфейсы между режимом пользователя и аппаратурой посредством так называемых системных вызовов (system calls), которые взаимодействуют с функциями ядра. Совокупность системных вызовов образует "интерфейс системных вызовов" (см. рис. 2.1).

Системные вызовы инициируют смену контекста выполнения процесса: процесс, работающий в режиме пользователя, переключается на выполнение в защищенном режиме (режим ядра). Такое переключение позволяет процессу вызывать защищенные процедуры ядра для выполнения системных функций. Таким образом, системные вызовы обеспечивают программный интерфейс для доступа к управлению системными ресурсами, такими как память, дисковое пространство и периферийные устройства. Системные вызовы реализованы в виде библиотеки времени выполнения (run-time library), а многие из них используются командными интерпретаторами shell. Следует заметить, что системные функции ядра для корректной работы требуют упаковки аргументов специальным образом.

В качестве примеров системных вызовов можно привести низкоуровневые функции ввода/вывода, такие как `open()`, `read()`, `write()` и `close()`.

Системные вызовы обеспечивают выполнение целого ряда операций:

- трансляции операций пользователя в запросы к драйверам устройств;
- создания, запуска и уничтожения процессов;
- ввода/вывода;
- доступа к файлам и дисковым устройствам;
- поддержки терминальных устройств.

Наличие интерфейсов в форме системных вызовов предоставляет определенные преимущества разработчикам программ. Во-первых, процесс разработки программ становится намного легче, поскольку программисту нет необходимости изучать особенности программных интерфейсов низкого уровня для каждого из устройств.

Во-вторых, повышается надежность системы в целом, поскольку ядро UNIX может проверить корректность запроса программы пользователя на уровне интерфейса, прежде чем ответить на такой запрос.

Наконец, наличие таких интерфейсов позволяет легко переносить программы на другие реализации UNIX.

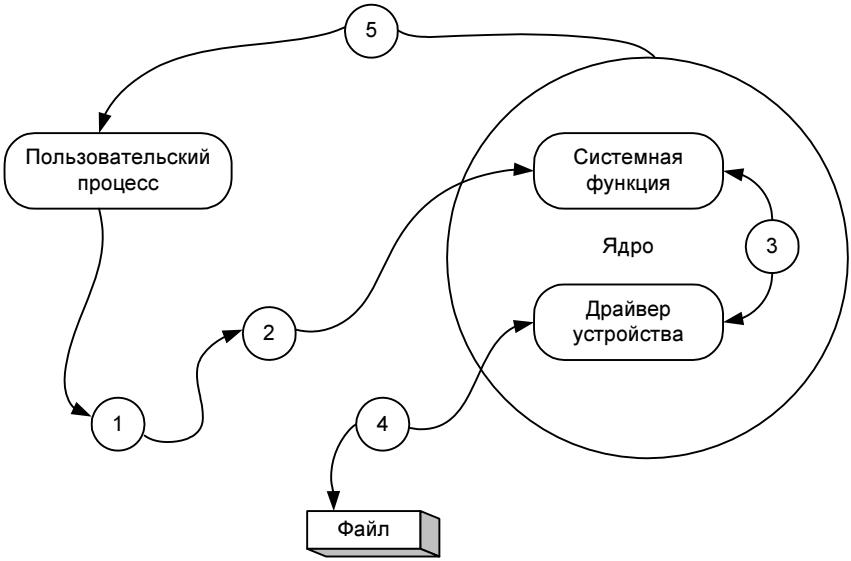


Рис. 2.4. Взаимодействие программы пользователя и ядра

Рассмотрим пример взаимодействия пользовательского процесса и ядра операционной системы UNIX. Предположим, что процессу пользователя необходимо открыть файл на диске для записи или чтения. В упрощенном виде последовательность шагов при открытии файла показана на рис. 2.4.

На шаге 1 процесс пользователя инициирует запрос на доступ к файлу (открытие файла). При этом программный код пользовательского процесса выполняет системный вызов при помощи функции `open()` (шаг 2). Далее (шаг 3) выполняется переключение контекста процесса из пользовательского режима в режим ядра и происходит обращение к системной функции ядра, соответствующей системному вызову `open()`.

Системная функция выполняет ряд операций, необходимых для открытия файла, находящегося на диске, например, формирует запросы к драйверу устройства (шаг 3) и анализирует статус (состояние) требуемого ресурса, полученный от драйвера. Драйвер устройства обращается к требуемому ресурсу и формирует статусную информацию и данные для системной функции (шаг 4).

На шаге 5 процесс пользователя получает от ядра результат выполнения запроса либо в виде определенных структур данных (для системного вызова `open()` это дескриптор открытого файла), либо сообщение об ошибке.

Системные вызовы в той или иной степени используются всеми без исключения пользовательскими и системными программами, и мы более подробно познакомимся с ними в *главе 12*.

Следующий уровень функциональности, который мы рассмотрим, — системные программы (см. рис. 2.1). К системным программам или, по-другому, к системному программному обеспечению относят командные оболочки shell (интерпретаторы), команды и утилиты системного администрирования, драйверы и протоколы коммуникаций. Как известно, операционная система UNIX включает ряд стандартных системных программ для выполнения задач администрирования, конфигурирования и поддержки файловой системы. Кроме того, к этой группе программ следует отнести утилиты:

- ❒ настройки параметров конфигурации системы;
- ❒ перекомпоновки ядра (если она необходима) и добавления новых драйверов устройств;
- ❒ создания и удаления учетных записей пользователей;
- ❒ создания и подключения физических файловых систем;
- ❒ установки параметров контроля доступа к файлам.

В качестве пользовательских программ могут выступать командные файлы, написанные с помощью командного интерпретатора shell, или разработанные на одном из языков высокого уровня (C, Pascal, Fortran) приложения. К пользовательским программам относятся многочисленные текстовые и графические редакторы, программы отправки и получения электронной почты и т. д. Следует отметить, что в некоторых случаях пользовательским программам не требуется обращение к функциям ядра. Процессы, порожденные пользовательскими программами, защищены от других пользовательских процессов, не имеют доступа к функциям ядра, кроме как через системные вызовы и, кроме того, не могут непосредственно обращаться к пространству памяти ядра.

Рассмотрим более подробно взаимодействие пользовательского процесса и ядра операционной системы.

Пространство памяти ядра представляет собой область памяти, в которой процессы ядра или, по-другому, процессы, работающие в контексте ядра, реализуют функции ядра. Пространство ядра — защищенная область, и пользователь получает к ней доступ только через интерфейс системных вызовов. Пользовательский процесс не имеет прямого доступа ко всем инструкциям и физическим устройствам — их имеет процесс ядра. Процесс ядра также может менять карту памяти, что необходимо для переключения процессов (смены контекста). Пользовательский процесс начинает работать в режиме ядра в тот момент, когда выполняется программный код ядра посредством системного вызова.

Поскольку пользовательские процессы и ядро не имеют общего адресного пространства памяти, необходим механизм передачи данных между ними.

При выполнении системного вызова аргументы вызова и соответствующий идентификатор процедуры ядра передаются из пользовательского пространства в пространство ядра. Идентификатор процедуры ядра передается либо через регистры процессора, либо через стек, а аргументы системного вызова передаются через область памяти вызывающего процесса. Область памяти пользовательского процесса содержит информацию о процессе, необходимую ядру, причем пользовательский процесс не может обращаться к пространству ядра, но ядро может обращаться к пространству процесса.

Одной из важнейших функций операционной системы является управление файловой системой. Файловая система представляет собой иерархически организованную структуру объектов, называемых файлами. Иногда выделяют группу объектов файловой системы, называемых каталогами, хотя каталог является разновидностью файла. Файловая система UNIX обеспечивает унифицированный интерфейс доступа к данным, расположенным на различных носителях, и к периферийным устройствам. Файловая система контролирует права доступа к файлам, выполняет операции создания и удаления файлов, а также осуществляет запись/чтение данных файла. Поскольку большинство прикладных функций выполняется через интерфейс файловой системы, следовательно, права доступа к файлам определяют привилегии пользователя в системе.

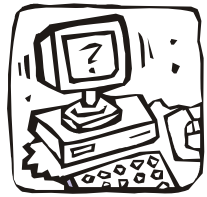
Особо следует отметить, что в операционной системе UNIX понятие "файл" трактуется в более широком смысле, чем в других операционных системах.

В общеупотребительном смысле под файлом понимают упорядоченный определенным образом набор данных в определенном формате (текст, графика, двоичные данные). В операционной системе UNIX к объектам файловой системы, помимо файлов, содержащих данные, также относятся и специальные файлы устройств (накопителей на жестких дисках, принтеров, сетевых интерфейсов и т. д.), а также сокеты и программные каналы.

Система управляет объектами файловой системы при помощи унифицированного набора системных функций ядра, к которым могут обращаться пользовательские процессы посредством системных вызовов, таких, например, как `open()`, `read()`, `write()` и `close()`. При этом все файловые операции, с точки зрения программы пользователя, выполняются одинаково как для файлов, содержащих данные, так и для специальных файлов устройств. Более подробно управление файловой системой мы рассмотрим в *главе 5*.

* * *

На этом анализ основных аспектов функционирования операционной системы UNIX можно закончить и приступить к более детальному рассмотрению отдельных подсистем.



Глава 3

Учетные записи пользователей

Для того чтобы пользователь мог работать в операционной системе, нужно сначала создать для него учетную запись (account). В процессе создания учетной записи пользователю присваивается имя, пароль, а также создается каталог, в который по умолчанию будет попадать пользователь после успешного входа в систему. Такой каталог часто называют домашним каталогом пользователя.

Любой пользователь операционной системы UNIX, помимо домашнего каталога, получает в свое распоряжение определенный набор ресурсов с установленными правами доступа к этим ресурсам. Например, пользователь может читать и записывать в файлы, выполнять печать документов на принтере, передавать и принимать сообщения электронной почты.

Для каждого пользователя в системе устанавливаются вполне определенные права (атрибуты) доступа к ресурсам, причем для разных пользователей они могут отличаться.

Все ресурсы операционной системы имеют определенные права доступа, причем часть таких прав устанавливается в процессе инсталляции операционной системы, остальные атрибуты доступа устанавливаются или изменяются специальным пользователем root, наделенным всеми правами по управлению (администрированию) операционной системы. Пользователь root имеет все права на подключение, удаление и модификацию учетных записей, назначение и изменение паролей, назначение кода доступа к ресурсам операционной системы.

3.1. Управление учетными записями

Далее мы рассмотрим более подробно, каким образом можно работать с учетными записями пользователей операционной системы, и начнем с регистрации пользователей.

Прежде чем пользователь начнет работу в системе, он должен зарегистрироваться в ней. Во время регистрации каждому пользователю назначается уникальный идентификатор сеанса и права доступа, позволяющие выполнять команды интерпретатора shell. В этот момент операционная система добавляет запись в файл, где хранятся записи о зарегистрированных пользователях.

Специальная программа, имеющая название `getty`, выдает на экран дисплея приглашение для регистрации пользователя. Для регистрации пользователь должен ввести специальный идентификатор (регистрационное имя), представляющий собой строку символов из цифр и букв алфавита. После ввода идентификатора процесс `getty` запускает программу регистрации `login`, которая использует в качестве параметра введенный идентификатор пользователя.

Программа `login` выполняет все необходимые действия по регистрации пользователя и установке параметров сеанса — процесса с уникальными идентификаторами пользователя и группы. Прежде всего, процесс `login` проверяет содержимое файла /etc/passwd, чтобы определить, требуется ли ввод пароля при входе в систему. Если требуется, то система выдает приглашение к вводу пароля на экран. Пароли большинства операционных систем в зашифрованном виде обычно хранятся в файле /etc/shadow. При успешной регистрации пользователя операционная система устанавливает параметры для текущего сеанса работы. Каждый сеанс пользователя получает числовой идентификатор пользователя и группы. Оба эти идентификатора определяют права доступа пользователя к объектам файловой системы, и без них невозможно выполнять ни одну команду.

Наконец, операционная система запускает командную оболочку (командный интерпретатор shell), которая позволяет выполнять команды.

Идентификация пользователей в операционной системе определяется несколькими файлами, имеющими ключевое значение для их работы: /etc/passwd, /etc/group и /etc/shadow. Информация о пользователе, сохраненная в этих файлах, позволяет системе устанавливать, изменять и выполнять другие функции управления учетной записью пользователя. Любой сеанс работы в UNIX начинается с регистрации пользователя.

Как было сказано, после регистрации пользователя система устанавливает его параметры для текущего сеанса работы. Идентификатор пользователя (UID) — это 32-разрядное целое число между 0 и 2 147 483 647. Значение 0 идентификатора присваивается суперпользователю root. Идентификаторы 1 и 2 присваиваются пользователям с административными правами (bin, daemon).

Рассмотрим более подробно, каким образом пользователю назначаются права доступа к файлам. Как известно, все объекты файловой системы имеют два идентификатора: пользователя и группы. Оба они наследуются от создавшего их процесса и определяют права доступа к файлу. Пользователь

и группа, идентификаторы которых связаны с файлом, считаются его владельцами. Изменить владельца файла можно с помощью команд `chown` и `chgrp`. Выделяют три категории доступа к объектам файловой системы:

- владелец файла (процесс, идентификатор пользователя которого равен идентификатору владельца файла);
- члены группы (процессы, идентификатор группы которых совпадает с идентификатором группы, установленным для файла);
- прочие — все остальные процессы.

Любому файлу при создании присваивается код доступа, представляющий собой комбинацию битов в индексном дескрипторе файла. Код доступа является комбинацией единичных значений битов, имеющих такой смысл:

- 000400 (100h) — разрешается чтение для владельца файла;
- 000200 (80h) — разрешается запись для владельца файла;
- 000100 (40h) — разрешается запуск на выполнение владельцу файла;
- 000040 (20h) — разрешается чтение членам группы;
- 000020 (10h) — разрешается запись членам группы;
- 000010 (8h) — разрешается выполнение членам группы;
- 000004 (4h) — разрешается чтение прочим пользователям;
- 000002 (2h) — разрешается запись прочим пользователям;
- 000001 (1h) — разрешается выполнение прочим пользователям.

Все коды показаны в восьмеричном формате, а в скобках представлены их шестнадцатеричные значения, которые в данной нотации условимся обозначать с завершающим символом "h".

Код доступа к файлу может быть изменен только привилегированным пользователем root. Нам уже встречался термин "привилегированный пользователь", поэтому дадим более точное его определение. Привилегированный пользователь — это пользователь, имеющий процесс с идентификатором, равным нулю. Независимо от кода защиты файла привилегированный пользователь имеет доступ ко всем файлам системы. Обычно в учетном файле пользователей /etc/passwd имеется привилегированный пользователь с именем root. Иногда вместо термина "привилегированный пользователь" используют другой — "суперпользователь".

Отдельные поля кода доступа предназначены для установки специальных атрибутов, позволяющих изменить идентификаторы пользователя и группы:

- 004000 (800h) — разрешение смены идентификатора пользователя при обращении к файлу (так называемый "Set-UID"-флаг);

- 002000 (400h) — разрешение смены идентификатора группы при обращении к файлу (так называемый "Set-GID"-флаг).

Зачем нужны такие атрибуты? Смысл состоит в том, что идентификаторы, полученные при запуске процесса ("реальные идентификаторы"), могут отличаться от идентификаторов, полученных после выполнения системного вызова `exec()` ("эффективные идентификаторы"). Изначально реальные и эффективные идентификаторы всегда совпадают, но если код доступа к файлу предусматривает смену идентификаторов, то после загрузки исполняемого файла с помощью системного вызова `exec()` реальные идентификаторы процесса изменяются на идентификаторы владельца (или группы) выполняемого файла, т. е. становятся эффективными.

Права доступа процесса проверяются по его эффективным идентификаторам. Если, например, владельцем файлов данных TEXT1, TEXT2 и TEXT3 является пользователь user, то полный доступ к этим файлам разрешен только ему. Предположим, что пользователю user принадлежат исполняемые файлы PROGRAM1, PROGRAM2 и PROGRAM3.

Пусть требуется, чтобы другие процессы, которые хотят выполнять операции над файлами TEXT1, TEXT2 и TEXT3, могли это сделать с помощью программ PROGRAM1, PROGRAM2 и PROGRAM3. Для этого в коде доступа файлов PROGRAM1, PROGRAM2 и PROGRAM3 необходимо установить биты разрешения смены идентификатора пользователя для процесса, вызвавшего на выполнение один из этих файлов. В этом случае другой процесс (не пользователя user), которому требуется доступ к одному из файлов TEXT1, TEXT2 или TEXT3, должен запустить через системный вызов `exec()` соответствующую программу обработки PROGRAM1, PROGRAM2 или PROGRAM3, иначе он вообще не получит доступа к защищенным их владельцем файлам данных TEXT1, TEXT2 или TEXT3.

Установку и изменение требуемых атрибутов доступа к файлу для пользователя, а также флагов Set-UID и Set-GID можно выполнить с помощью системного вызова `chmod()`. Можно воспользоваться также командой `chmod`, которая представляет собой "обертку" соответствующего системного вызова и доступна из командной оболочки shell. Более подробно установку и изменение атрибутов защиты объектов файловой системы мы рассмотрим в *главе 5*.

Информацию о реальных и эффективных идентификаторах выполняющегося процесса можно получить с помощью системных вызовов `getuid()` и `getgid()`. Выполняющийся процесс может динамически изменять свои идентификаторы с помощью системных вызовов `setuid()` и `setgid()`, однако такое допускается только для привилегированного процесса (выполняющегося с идентификатором root) или такого, у которого реальный идентификатор совпадает с устанавливаемым.

Например, специальная программа `passwd` позволяет изменить пароль пользователя в учетном файле пользователей /etc/passwd. Владельцем этих файлов является суперпользователь root. Код доступа файла /etc/passwd разрешает выполнять запись данных только владельцу, в то время как код доступа исполняемого файла `passwd` разрешает смену пароля пользователя. Следовательно, пользователь, отличный от root, может изменить свой пароль только с помощью программы `passwd`, что является корректным по отношению к пользователю, который всегда может изменить свой пароль, не сообщая об этом системному администратору.

Ключевую роль в регистрации и управлении учетными записями пользователей играют несколько файлов, упоминавшихся ранее в этой главе: /etc/passwd, /etc/group и /etc/shadow. Рассмотрим их более подробно и начнем с анализа формата файла паролей passwd, на котором основано применение других форматов подобных файлов паролей. Этот файл обычно расположен в каталоге /etc и состоит из однострочных записей, представляющих собой отдельные учетные записи. Поля каждой записи отделены друг от друга двоеточием.

Файл passwd имеет одинаковую структуру во всех операционных системах UNIX. Вот пример записи файла /etc/passwd для пользователя yury:

```
$ cat /etc/passwd|grep yury
yury:x:500:500:Yury:/home/yury:/bin/bash
$
```

Все записи имеют семь полей, разделенных двоеточием. Приведем расшифровку (слева направо) отдельных полей, в качестве примера используя показанную выше запись:

- поле 1 — регистрационное имя пользователя (`yury`);
- поле 2 — зашифрованный пароль (`x`);
- поле 3 — идентификатор пользователя (`500`);
- поле 4 — идентификатор группы (`500`);
- поле 5 — информация о пользователе (`Yury`);
- поле 6 — рабочий каталог пользователя (`/home/yury`);
- поле 7 — используемая командная оболочка (`/bin/bash`).

Для работы в многопользовательских системах UNIX имеет механизм группового доступа. Несмотря на то, что учетная запись пользователя может входить в одну или несколько групп, принадлежать она может только одной группе. Для этого в файле паролей /etc/passwd присутствует поле идентификатора группы (group ID, GID).

Информация о принадлежности пользователей к тем или иным группам находится в файле /etc/group. Чтобы сделать пользователя членом какой-либо группы, необходимо внести соответствующие коррективы в файл /etc/group. Следует заметить, что в ранних версиях операционной системы UNIX пользователь мог входить только в одну группу. Для современных систем такое ограничение снято, и пользователь может являться членом 16 групп одновременно. Поле идентификатора группы в настоящее время практически не используется операционной системой, тем не менее, ему присваивается определенное значение.

Рассмотрим теперь, как учетные записи пользователей создаются, изменяются и удаляются. Для создания, изменения и удаления учетных записей в операционных системах UNIX имеется специальная группа команд. Вначале остановимся на командах, используемых в операционных системах, совместимых с System V:

- `useradd` — используется для создания новой учетной записи пользователя или изменения информации о пользователе;
- `userdel` — служит для удаления регистрационного имени пользователя из системы;
- `passwd` — позволяет изменить пароль пользователя;
- `su` — выполняет команду с заменой идентификатора пользователя и группы;
- `login` — инициализирует сессию пользователя в системе;
- `id` — отображает реальный и эффективный идентификаторы пользователя и группы;
- `pwconv`, `pwunconv`, `grpconv`, `grpunconv` — выполняет преобразование обычных и теневых файлов паролей и групп;
- `pwck` — проверяет целостность файла паролей.

Проанализируем эти команды более подробно и начнем с команды `useradd`. Вызванная без опции `-D`, команда `useradd` создает новый бюджет пользователя, используя параметры командной строки. Остальные параметры принимаются по умолчанию. После выполнения команды новая учетная запись пользователя будет зарегистрирована в системе, будет создан домашний каталог пользователя, куда копируются файлы инициализации.

Хочу напомнить, что создание, удаление и управление учетными записями может выполнять только суперпользователь root. Рассмотрим некоторые, наиболее часто используемые опции команды `useradd`:

- `-u` *`идентификатор`* — указывает идентификационный номер пользователя (UID). Этот номер должен представлять собой неотрицательное целое

число, меньшее по значению, чем системный параметр MAXUID. Если номер не указан, то по умолчанию будет использован следующий доступный UID. Например, если в системе уже используются UID с номерами от 100 до 105, то следующий UID будет равен 106. Следует отметить, что идентификаторы с номерами 0—99 зарезервированы системой и использоваться не могут;

- `-o` — эта опция позволяет создать дубликат UID. Использовать ее следует крайне осторожно, поскольку усложняется обеспечение безопасности системы в целом из-за наличия неоднозначности соответствия UID определенному пользователю;
- `-g` *группа* — представляет собой целочисленный идентификатор или символьное имя существующей группы. Она определяет основную (primary) группу для нового пользователя;
- `-G` *группа* — представляет собой несколько элементов списка, разделенных запятыми, каждый из которых является целочисленным идентификатором или символьным именем существующей группы, при этом список может состоять из одного элемента. Содержимое списка устанавливает принадлежность пользователя к дополнительным группам, которые могут быть определены с помощью команды `newgrp`;
- `-d` *каталог* — начальный (домашний) каталог нового пользователя. По умолчанию в качестве начального используется каталог HOME/registration_name, где HOME — базовый каталог для начальных каталогов новых пользователей, а registration_name — регистрационное имя нового пользователя;
- `-s shell` — полный путь к командному интерпретатору, используемому пользователем сразу же после регистрации. По умолчанию этому полю значение не присваивается, поэтому система использует стандартный командный интерпретатор /usr/bin/sh. Для командной оболочки shell нужно указывать существующий исполняемый файл;
- `-c` *комментарий* — любая текстовая строка, кратко описывающая регистрационное имя (обычно указывает фамилию и имя реального пользователя). Эта информация хранится в записи пользователя в файле /etc/passwd. Размер данного поля не должен превосходить 128 символов;
- `-m` — создает домашний каталог для нового пользователя, если таковой еще не существует. Если каталог уже существует, вновь созданный пользователь должен обладать правами доступа к указанному каталогу;
- `-k skel_dir` — выполняет копирование содержимого каталога skel_dir в начальный каталог нового пользователя вместо использования стан-

дартного "шаблонного" каталога /etc/skel, содержащего стандартные файлы, определяющие среду работы пользователя. Каталог skel_dir должен существовать до выполнения операции;

- `-f` *активно_дней* — максимально допустимый интервал времени в днях между использованиями регистрационного имени пользователя, когда это имя еще не объявляется недействительным. Обычно в качестве значений указываются положительные целые числа;
- `-e` *дата* — дата, начиная с которой регистрационное имя пользователя нельзя будет использовать: после этой даты ни один пользователь не сможет войти в систему, указывая данное регистрационное имя;
- `login` — строка символов, задающая регистрационное имя для нового пользователя. В ней не должны присутствовать символы двоеточия и перевода строки, а первый символ не должен быть прописной буквой.

Рассмотрим пример создания учетной записи пользователя с регистрационным именем alex. Это будет работать в операционных системах Solaris и Linux. Для создания учетной записи пользователя необходимо зарегистрироваться в системе как суперпользователь root. Перед созданием учетной записи желательно просмотреть опции по умолчанию для команды `useradd`. Они могут быть, например, такими:

```
# useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/bash
SKEL=/etc/skel
#
```

По результату выполнения команды `useradd -D` можно сделать несколько важных выводов. Во-первых, в качестве корневого каталога для вновь создаваемых пользователей используется каталог /home. Во-вторых, пустое поле значения параметра `EXPIRE` означает, что учетная запись пользователя никогда не будет заблокирована. Наконец, в качестве командного интерпретатора для всех вновь создаваемых пользователей по умолчанию установлен /bin/bash.

Создадим учетную запись пользователя alex. Для этого введем команду `useradd`, в которой зададим регистрационное имя:

```
# useradd alex
```

Если команда выполнена успешно, то пользователь alex будет зарегистрирован в системе, и в файл /etc/passwd будет добавлена запись, идентифицирующая этого пользователя. Просмотреть ее можно с помощью команды `cat`:

```
# cat /etc/passwd|grep alex
alex:x:2307:2307::/home/alex:/bin/bash
#
```

Команда `useradd` автоматизирует процесс регистрации пользователя, но можно сделать это вручную, если нужно установить какие-либо индивидуальные параметры для пользователя, например, командную оболочку или домашний каталог. Процедура не очень сложна и требует выполнения нескольких шагов, перечисленных далее.

1. Предположим, необходимо создать учетную запись пользователя с регистрационным именем newuser. Вначале просмотрим файл /etc/passwd/ на предмет поиска наибольшего UID. Наибольшее значение UID (2307) имеет вновь созданный пользователь alex, поэтому следующим UID может быть 2308. Возьмем это на заметку и добавим в файл /etc/passwd запись о пользователе newuser:

   ```
   # echo newuser:x:2308:2308::/home/newuser:/bin/bash >> /etc/passwd
   ```

2. Далее создадим начальный каталог пользователя newuser, выбрав путь к /home/newuser:

   ```
   #mkdir /home/newuser
   ```

3. Делаем пользователя newuser владельцем каталога /home/newuser:

   ```
   # chown newuser newuser
   ```

4. Приводим в соответствие записи файлов /etc/passwd и /etc/shadow с помощью команды `pwconv`:

   ```
   #pwconv
   ```

Команда `pwconv` создает файл shadow из passwd, при этом может использоваться и существующий файл shadow (он будет перезаписан). Команда работает следующим образом: сначала удаляются те записи в теневом файле shadow, которые не существуют в основном файле паролей passwd. Далее обновляются те теневые записи, для которых в полях пароля в основном файле не стоит "x", и добавляются все недостающие теневые записи. После этого пароли в основном файле заполняются символами "x".

Удалить учетную запись пользователя в операционных системах System V можно с помощью команды `userdel`. Эта команда удаляет информацию

о пользователе из системы, выполняя соответствующие изменения в регистрационных файлах и файловой системе. Кроме этого, команда запоминает идентификационный номер удаляемого пользователя (UID) в файле /etc/security/ia/ageduid для того, чтобы этот идентификатор не использовался повторно в течение определенного периода времени. Такой механизм называется устареванием идентификатора (UID aging).

Команда имеет синтаксис:

```
userdel [-r] [-n_месяцев] имя
```

Опции команды имеют следующий смысл:

- `-r` — удаление начального каталога пользователя из системы, при этом каталог должен существовать. Если команда выполнена успешно, файлы и подкаталоги в начальном каталоге будут недоступны;
- `-n_месяцев` — задает интервал времени в месяцах, указывающий, как долго идентификатор пользователя должен устаревать перед повторным использованием. Если параметр равен –1, то идентификатор пользователя никогда не будет повторно использован, если он равен 0, то идентификатор пользователя можно использовать немедленно. Если опция `-n` не задана, принимается значение устаревания по умолчанию.

Изменить параметры учетной записи пользователя в системах System V можно при помощи команды `usermod`. Эта команда модифицирует файлы, содержащие информацию об учетных записях пользователей. Допустимы следующие опции:

- `-A метод|DEFAULT` — указывает новый метод идентификации пользователя и представляет собой имя программы, отвечающей за допустимую идентификацию пользователя. Строку `DEFAULT` можно использовать для установки стандартного метода идентификации;
- `-c комментарий` — указывает на другой комментарий для записи пользователя в файле паролей;
- `-d домашний_каталог` — новый домашний каталог пользователя. При указании опции `-m` содержимое текущего домашнего каталога будет перемещено в новый домашний каталог, который будет создан, если еще не существует;
- `-e дата` — дата, после которой учетная запись пользователя устареет. Дата указывается в формате `MM/DD/YY`;
- `-f активно_дней` — число дней между датой устаревания пароля и датой, когда учетная запись пользователя будет заблокирована. Значение, равное 0,

блокирует учетную запись пользователя в момент устаревания пароля, а значение –1 запрещает блокировку (значение по умолчанию);

- `-g` *группа* — имя группы или номер группы, которая будет присвоена пользователю после входа в систему, причем группа с указанным именем должна существовать. Номер группы также должен ссылаться на существующую группу (по умолчанию равен 1);
- `-G` *дополнительная_группа* — список дополнительных групп. Данный пользователь также является членом этих групп. Каждая группа отделяется от следующей группы запятой, без пробелов. Группы являются предметом для некоторых ограничений, например, группа, заданная с опцией `-g`. Если пользователь является членом группы, которая отсутствует в списке, то пользователь будет удален из группы;
- `-l` *новое_имя* — имя пользователя будет изменено с *имя* на *новое_имя*. Ничего другого сделано не будет. В частности, домашний каталог пользователя должен быть, вероятно, изменен;
- `-s shell` — имя командного интерпретатора, который будет использоваться новым пользователем при входе в систему. Установка этого поля в пустое значение означает выбор системного shell;
- `-u uid` — числовое значение идентификатора пользователя (UID). Это значение должно быть уникальным, исключение составляет использование опции `-o`. Значение должно быть положительным. Как было сказано ранее, значения между 0 и 99 обычно зарезервированы для системных бюджетов. Для любых файлов, владельцем которых является пользователь и которые находятся в домашнем каталоге пользователя, идентификатор пользователя (UID) будет изменяться автоматически. Для файлов вне домашнего каталога пользователя идентификатор пользователя должен быть изменен вручную.

Замечание

Если пользователь находится в системе, изменить его имя не удастся.

Рассмотрим практический пример использования команды `usermod`. Ранее мы создали учетную запись пользователя newuser. Изменим регистрационное имя пользователя и домашний каталог. Напомню, как выглядит запись для пользователя newuser в файле /etc/passwd:

```
# cat /etc/passwd|grep newuser
newuser:x:2308:2308::/home/newuser:/bin/bash
#
```

Присвоим пользователю newuser регистрационное имя moduser и изменим домашний каталог на /home/moduser:

```
# usermod -l moduser -d /home/moduser -m newuser
#
```

Можно проверить, что изменения выполнены успешно:

```
# cat /etc/passwd|grep moduser
moduser:x:2308:2308::/home/moduser:/bin/bash
#
```

Проверить наличие домашнего каталога пользователя moduser можно, задав команду:

```
[root@localhost root]# ls -l /home
total 24
drwxr-xr-x   10 moduser  root         4096 Aug  18 22:51 moduser
drwx------   26 yury     yury         4096 Aug  18 00:24 yury
[root@localhost root]#
```

Здесь нужно сделать одно важное замечание: при изменении регистрационного имени пользователя его UID остается неизменным. Это свидетельствует о том, что операционная система работает с одной и той же учетной записью пользователя, несмотря на то, что регистрационное имя пользователя изменилось. Таким образом, можно сделать очень важный вывод: для UNIX определяющим фактором при работе с пользователем является его идентификатор (UID).

Еще одной командой, которую мы проанализируем, является `passwd` — она позволяет изменить пароли пользователей. Обычные пользователи могут изменить пароль только для своей учетной записи, в то время как суперпользователь root может изменять пароли всех пользователей.

Команда `passwd` также позволяет изменить информацию об учетной записи: полное имя пользователя, его командный интерпретатор, дату истечения срока используемого пароля и интервал времени, в течение которого пароль действует.

Команда имеет синтаксис:

```
passwd [-f] имя
passwd [-g] [-r|R] группа
passwd [-x  max] [-n min] [-w warn] [-i inact] имя
passwd {-l|-u|-d|-S} имя
```

Если пользователь имеет пароль, то перед установкой нового пароля команда `passwd` предлагает ввести старый пароль, который хранится в зашифрован-

ном виде. Пользователь может ввести правильный пароль только один раз, хотя суперпользователь root может пропустить этот шаг, поэтому если пароль забыт, то его все же можно изменить. После введения пароля команда проверяет наличие разрешения на изменение пароля в данное время. Если это не разрешено, программа `passwd` завершает свою работу без изменения пароля.

Рассмотрим использование некоторых опций команды `passwd` более подробно.

- `-g` — замена пароля для заданной группы. Эта опция требует наличия прав суперпользователя или администратора заданной группы. Опция `-r` применяется совместно с опцией `-g` для удаления текущего пароля заданной группы, что делает группу доступной всем членам. Опция `-R` используется совместно с опцией `-g` для ограничения доступа к группе всем пользователям.
- Информация об истечении срока действия пароля может быть изменена суперпользователем с помощью опций `-x`, `-n`, `-w` и `-i`. Опция `-x` нужна для установки максимального числа дней, в течение которых пароль остается допустимым, при этом после `max` дней требуется его изменение. Опция `-n` служит для установки минимального числа дней, после истечения которых пароль может быть изменен. Пользователю запрещается изменять пароль в течение `min` дней. Опция `-w` предназначена для установки числа дней, в течение которых пользователь будет получать предупреждающее сообщение об истечении времени действия его пароля, причем сообщения будут выводиться в течение `warn` дней, напоминая пользователю, сколько дней осталось до момента устаревания его пароля. Опция `-i` запрещает использование учетной записи пользователя по истечении промежутка времени после устаревания пароля. Если устаревший пароль остается неизменным в течение `inact` дней, он не будет вновь принят системой.
- Обслуживание учетной записи. Учетные записи пользователей могут быть заблокированы и разблокированы при помощи опций `-l` и `-u`. Опция `-l` блокирует учетную запись, изменяя пароль таким образом, что он становится непригодным для шифрования. Опция `-u` разблокирует учетную запись, изменяя пароль к его предыдущему значению.
- Статус учетной записи может быть получен с помощью использования опции `-S`. Статусная информация состоит из шести полей. Первое поле кодируется следующим образом: `L` — если бюджет пользователя заблокирован, `NP` — если не существует пароля для данной учетной записи и `P` — если пароль используется. Во втором поле указана дата последнего изменения пароля. Следующие четыре поля — минимальное время до истечения срока действия пароля, максимальное время до истечения срока дей-

ствия пароля, период вывода предупреждающего сообщения об истечении срока действия пароля и период неактивности для этого пароля.

- Опция `-f` требует от пользователя изменить пароль при следующем входе в систему.

При выборе пароля можно руководствоваться следующими соображениями. Безопасность пароля зависит в первую очередь от применяемого алгоритма шифрования и размера ключа. В операционных системах UNIX метод криптографии основывается на алгоритме NBS DES, который имеет очень высокую степень безопасности, при этом размер ключа зависит от выбранного пароля. Еще одно эмпирическое правило — лучше не выбирать пароль, в котором используются литературные выражения. Нередко пользователь выбирает пароль, основанный на личных данных (месяц, год рождения и т. д.). Подобный пароль расшифровывается довольно легко, поэтому такой "алгоритм шифрования" не годится.

Вместе с тем пароль должен хорошо запоминаться, поэтому неплохим вариантом в этом смысле является пароль, представляющий собой знакомое слово, части которого разделены специальными символами, или законченное слово, состоящее из двух слов, объединенных вместе и разделенных специальными символами или цифрами. Примерами таких паролей являются Pe&ter$sbu)rg, my!Pas#s%word. Еще одним методом составления паролей является комбинирование первых символов какой-либо фразы из литературного произведения. Воспользуемся, например, для создания пароля фразой

```
Astala vista
```

Пароль может быть таким:

```
As79tal01avi&sta
```

Это достаточно бессмысленный пароль, который, однако, несложно запомнить.

Рассмотрим механизм управления учетными записями пользователей в BSD-совместимых операционных системах на примере FreeBSD. Здесь для хранения данных, относящихся к учетным записям, используются файлы /etc/passwd и /etc/master.passwd, при этом файл /etc/master.passwd в некотором смысле напоминает /etc/shadow, используемый в системах System V.

Файл /etc/master.passwd хранит ту же информацию, что и /etc/passwd, хотя есть и некоторые отличия. Так, например, в этом файле хранятся хеш-коды (шифры) пользовательских паролей и целый ряд других сведений. Файл /etc/master.passwd содержит в зашифрованном виде пароли пользователей, поэтому он доступен только для чтения суперпользователю root.

В операционной системе FreeBSD, как и в других системах, имеются команды для создания, изменения и удаления учетных записей пользователей.

Для создания учетных записей пользователей применяется утилита `adduser`. При первом вызове программа `adduser` пытается обратиться к конфигурационному файлу /etc/adduser.conf, который в стандартной конфигурации отсутствует. Во время работы эта программа выводит подсказки с предлагаемыми настройками, причем для принятия установки, предлагаемой программой, достаточно нажать клавишу <Enter>.

Следует сказать, что синтаксис команды `adduser` во многом напоминает синтаксис команды `useradd`.

Другая команда, `chpass`, применяется для изменения учетных записей пользователей в FreeBSD. С ее помощью можно изменять параметры учетной записи, включая пароль, срок действия учетной записи и стандартный интерпретатор команд. Синтаксис этой команды таков:

```
chpass [-a список] [-p зашифрованный_пароль] [-e срок_действия]
       [-s интерпретатор] [login]
```

Опции команды `chpass` имеют следующий смысл:

- `-a` *`список`* — позволяет суперпользователю определять полную запись в формате /etc/passwd;
- `-p` *`зашифрованный_пароль`* — разрешает менять пароль, предварительно зашифрованный командой `crypt`. Опция обычно используется в командных сценариях, содержащих команду `crypt` и передающих полученный результат команде `chpass`;
- `-e` *`срок_действия`* — задает срок действия учетной записи;
- `-s` *`интерпретатор`* — обеспечивает смену стандартного интерпретатора команд на указанный;
- `login` — задает модифицируемую учетную запись.

В большинстве случаев команда `chpass` используется без параметров или с единственным параметром `login`. В таких случаях запускается редактор, в котором можно изменить параметры учетной записи.

Учетные записи пользователей FreeBSD можно отредактировать вручную непосредственно в файле /etc/master.passwd, после чего распространить изменения на другие файлы с помощью команды `pwd_mkdb`:

```
#pwd_mkdb -p /etc/master_passwd
```

Рассмотрим еще одну команду — `rmuser`. Команда имеет простой синтаксис:

```
rmuser [-y] login
```

и выполняет следующие действия:

- уничтожает процессы, инициированные пользователем;
- удаляет задания демона `cron`, запланированные пользователем;
- удаляет задания команды `at`, запланированные пользователем;
- удаляет относящиеся к пользователю записи из файлов паролей (/etc/passwd, /etc/master.passwd);
- удаляет почтовую очередь пользователя из каталога /var/mail;
- удаляет файлы пользователя из каталогов /tmp, /var/tmp, /var/tmp/vi.recover;
- удаляет учетную запись пользователя из всех групп в файле /etc/group и саму группу, если пользователь является ее единственным членом;
- интерактивно позволяет удалить начальный каталог пользователя.

Группы пользователей в системе FreeBSD можно создавать либо с помощью программы `sysinstall`, либо вручную, редактируя файл /etc/group. Как видно из обзора команды управления учетными записями пользователей, System V и FreeBSD очень похожи и используют однотипные параметры. Мы рассмотрели только основные команды для управления учетными записями пользователей, хотя имеется целый ряд других команд, выполняющих более узкие задачи управления учетными записями. Более подробную информацию о них можно получить из man-страниц.

3.2. Программный интерфейс управления пользователями

Все операционные системы UNIX, помимо стандартных команд управления пользователями, доступных из интерпретатора shell, имеют программные интерфейсы языков высокого уровня (C, Pascal, Fortran и т. д.), с помощью которых можно реализовать свои, довольно изощренные алгоритмы управления учетными записями пользователей, получать различного рода информацию о пользователях.

Такие интерфейсы значительно расширяют возможности командных оболочек, поскольку с их помощью можно выполнять довольно сложные и комплексные задачи по управлению и настройке операционной системы. Это касается не только управления учетными записями пользователей, но и других операций, например, манипуляций объектами файловой системы, управления процессами, обработки электронной почты и т. д.

Кроме того, пользователь может создавать специализированные программы, включающие в себя часть функций операционной системы. Например, при

разработке приложений, требующих обработки файлов данных, понадобится написать программный код для разграничения доступа пользователей к таким файлам. В этом случае, да и во многих других, знание программного интерфейса языков высокого уровня может оказать неоценимую помощь.

Наиболее мощным и популярным языком программирования в UNIX-системах является C. При этом следует учитывать и тот факт, что сама операционная система UNIX написана на языке C, поэтому профессиональная работа в этой системе требует неплохого знания этого языка.

В этой и последующих главах мы будем часто использовать язык C для создания программ, расширяющих возможности пользователя по управлению операционной системой. Прежде чем мы начнем создавать программы, я хочу обратить внимание читателей на ряд важных моментов. Первый касается интерфейса прикладного программирования (Application Programming Interface, API) операционных систем UNIX.

В UNIX-системах имеется набор функций API, более известный под названием "системные вызовы", которые используются программами пользователей и большинством системных утилит для выполнения определенных системных задач. Такие задачи нельзя решить, используя только стандартные библиотечные функции языка C. Более того, многие команды UNIX и библиотечные функции C используют API-функции в процессе своей работы. Таким образом, применяя интерфейс прикладного программирования UNIX, можно избежать издержек, связанных с использованием библиотечных функций C.

Большинство UNIX-систем включает набор API-функций, позволяющих выполнять такие задачи:

- получение данных о настройках системы;
- получение информации о пользователях;
- манипуляции объектами файловой системы;
- управление процессами;
- обеспечение межпроцессного взаимодействия;
- обеспечение сетевого взаимодействия.

Большинство API-функций операционной системы UNIX обращается к внутренним ресурсам ядра. Если, например, процесс обращается к API-функции ("системному вызову"), то контекст выполнения переключается из режима пользователя в режим ядра (об этом упоминалось в *главе 2*). Вызов API-функции в общем случае требует больше времени, чем вызов пользовательской функции, именно из-за переключения контекста, поэтому при разработке быстродействующих программ следует по возможности избегать системных вызовов.

Второй важный момент, на котором я хочу остановиться, касается использования библиотечных функций С. В языке С определен набор библиотечных функций, которые не имеют прямого соответствия в интерфейсах прикладного программирования (API), но, тем не менее, довольно широко используются при разработке программ для UNIX. Особое внимание хочу обратить на то, что в С имеется целый ряд библиотечных функций, не определенных в стандарте ANSI С, но, тем не менее, доступных во всех UNIX-системах.

Таким образом, при разработке эффективных программ пользователь должен учитывать особенности использования как библиотечных функций С, так и API-функций операционной системы UNIX.

В примерах программ, представленных в этой и последующих главах, мы будем использовать как системные вызовы UNIX, так и библиотечные функции С. Программы, представленные в этой и последующих главах, будут работать во всех наиболее популярных операционных системах (Linux, FreeBSD, Solaris и т. д.). Компиляцию исходных текстов программ можно выполнить стандартными средствами, входящими в состав UNIX. Например, в операционной системе Linux, равно как и в других системах, можно использовать популярный пакет g++ компилятора С со стандартными опциями.

Более подробно возможности операционной системы UNIX по разработке программного обеспечения будут проанализированы в *главе 12*, а сейчас рассмотрим некоторые примеры программ на языке С, демонстрирующие манипуляции с учетными записями пользователей.

Пример 1. Программа (назовем ее `home_dir`) отображает на экране дисплея домашний каталог пользователя, чье имя указано в качестве аргумента программы. Исходный текст программы (home_dir.c) показан далее:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>

int main(int argc, char* argv[])
{
 struct passwd *pwd;
 if (argc != 2)
 {
  printf("Usage: %s user_name\n", argv[1]);
  exit(0);
 }
```

```
 pwd = getpwnam(argv[1]);
 if (!pwd)
 {
  printf("%s is not a valid user name!\n", argv[1]);
  exit(1);
 }
 printf("Home directory for user '%s': %s\n", argv[1], pwd->pw_dir);
 return 0;
}
```

В заголовке `<pwd.h>` определяется набор функций, предназначенных для получения информации об учетной записи пользователя, содержащейся в файле /etc/passwd. Кроме того, здесь же определена структура `passwd`, в поля которой помещается информация из файла /etc/passwd. Структура имеет формат:

```
struct passwd
 {
  char*     pw_name     // регистрационное имя пользователя
  char*     pw_passwd   // зашифрованный пароль
  int       pw_uid      // идентификатор (UID) пользователя
  int       pw_gid      // идентификатор (GID) группы
  char*     pw_age      // минимальный срок действия пароля
  char*     pw_comment  // общая информация о пользователе
  char*     pw_dir      // начальный каталог пользователя
  char*     pw_shell    // регистрационная командная оболочка (shell)
}
```

В данном примере используется функция `getpwnam()`, имеющая синтаксис:

```
const struct passwd* getpwnam(const char* имя_пользователя)
```

Здесь *имя_пользователя* — регистрационное имя пользователя.

Результат выполнения программы для различных пользователей (в том числе и несуществующих) показан далее:

```
root@yurylinx ]# ./home_dir root
Home directory for user 'root': /root
[root@yurylinx ]# ./home_dir yury
Home directory for user 'yury': /home/yury
[root@yurylinx ]# ./home_dir yuri
yuri is not a valid user name!
```

Пример 2. В этом примере на экран дисплея выводится имя пользователя и путь к регистрационному командному интерпретатору для заданного идентификатора (UID) пользователя, который является единственным параметром программы. Программа называется `get_name_shell`, а исходный текст, сохраненный в файле get_name_shell.c, показан далее:

```
int main(int argc, char* argv[])
{
 struct passwd *pwd;
 int uid;

 if (argc != 2)
 {
  printf("Usage: %s UID\n", argv[0]);
  exit(0);
 }
 uid = atoi(argv[1]);
 pwd = getpwuid(uid);

 if (!pwd)
 {
  printf("%s is not a valid user UID!\n",argv[1]);
  exit(1);
 }
 //
 printf("SHELL for user with UID = %d is: '%s'\n", pwd->pw_uid,
 pwd->pw_shell);
 printf("USER_NAME for user with UID = %d is: '%s'\n", pwd->pw_uid,
 pwd->pw_name);
 return 0;
}
```

В этой программе используется функция `getpwuid()`, синтаксис которой таков:

```
const struct passwd* getpwuid(const int uid)
```

Здесь `uid` — идентификатор пользователя. Результат работы программы `get_name_shell` показан далее:

```
[root@yurylinx ]# ./get_name_shell 0
SHELL for user with UID = 0 is: '/bin/bash'
USER_NAME for user with UID = 0 is: 'root'
```

```
[root@yurylinx ]# ./get_name_shell 500
SHELL for user with UID = 500 is: '/bin/bash'
USER_NAME for user with UID = 500 is: 'yury'
[root@yurylinx ]# ./get_name_shell 2
SHELL for user with UID = 2 is: '/sbin/nologin'
USER_NAME for user with UID = 2 is: 'daemon'
```

Пример 3. Программа `get_all_names_uids`, исходный текст которой находится в файле get_all_names_uids.c, выводит на экран дисплея имена и идентификаторы пользователей, записи о которых находятся в файле /etc/passwd.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>

int main(void)
{
 struct passwd *pwd;
 setpwent();
 while(pwd = getpwent())
  {
   printf("USER_NAME:%s, UID:%d\n", pwd->pw_name, pwd->pw_uid);
  }
 endpwent();
 return 0;
}
```

В этой программе используются функции `setpwent()`, `getpwent()` и `endpwent()`. Функция `setpwent()` устанавливает указатель чтения на начало файла /etc/passwd, функция `getpwent()` смещает указатель на следующую запись файла /etc/passwd. Для закрытия файла /etc/passwd вызывается функция `endpwent()`.

Результат выполнения программы `get_all_names_uids` показан далее:

```
[root@yurylinx ]# ./get_all_names_uids
USER_NAME:root, UID:0
USER_NAME:bin, UID:1
USER_NAME:daemon, UID:2
USER_NAME:adm, UID:3
USER_NAME:lp, UID:4
```

```
. . .
USER_NAME:nobody, UID:99
USER_NAME:rpm, UID:37
USER_NAME:xfs, UID:43
USER_NAME:named, UID:25
. . .
USER_NAME:yury, UID:500
```

Пример 4. С помощью программы `get_grp_id`, исходный текст которой показан далее, можно отобразить на экране идентификатор группы, имя которой задается в качестве параметра программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <grp.h>

int main(int argc, char* argv[])
{
 struct group *grp;
 if (argc != 2)
 {
  printf("Usage: %s [group_name]\n", argv[0]);
  exit(0);
 }
 grp = getgrnam(argv[1]);
 printf("GID = %d for GROUP %s\n", grp->gr_gid, grp->gr_name);
 return 0;
}
```

В заголовке `<grp.h>` определяется набор функций, предназначенных для получения информации о группах, содержащейся в файле /etc/group. Кроме того, здесь же определена структура `group`, в поля которой помещается информация из файла /etc/group. Структура имеет формат:

```
struct group
 {
  char*     gr_name     // имя группы
  char*     gr_passwd   // зашифрованный пароль группы
  int       gr_gid      // идентификатор (GID) группы
  char*     pw_comment  // имена членов группы
}
```

Функция `getgrnam()` принимает в качестве аргумента имя группы и возвращает указатель на запись типа `struct group`, которая содержит информацию о группе, если группа определена в системе.

Далее показан результат работы программы `get_grp_id`:

```
[root@yurylinx ]# ./get_gr_id yury
GID = 500 for GROUP yury
[root@yurylinx ]# ./get_gr_id daemon
GID = 2 for GROUP daemon
```

Пример 5. Программа `show_grp`, исходный текст которой показан далее, отображает на экране информацию о группах (имя группы и идентификатор), содержащуюся в файле /etc/group:

```
#include <stdio.h>
#include <stdlib.h>
#include <grp.h>

int main(void)
{
 struct group *grp;
 setgrent();
 while (grp = getgrent())
 {
  printf("GROUP_NAME: %s, GID: %d\n", grp->gr_name, grp->gr_gid);
 }
 endgrent();
 return 0;
}
```

В этой программе функция `setgrent()` устанавливает указатель чтения файла на начало файла /etc/group, функция `getgrent()` смещает на следующую запись файла /etc/group, а функция `endgrent()` закрывает файл /etc/group.

Результат работы программы `show_grp` показан далее:

```
[root@yurylinx ]# ./show_grp
GROUP_NAME: root, GID: 0
GROUP_NAME: bin, GID: 1
GROUP_NAME: daemon, GID: 2
GROUP_NAME: sys, GID: 3
. . .
```

```
GROUP_NAME: desktop, GID: 80
GROUP_NAME: yury, GID: 500
```

Рассмотренные примеры демонстрируют только небольшую часть тех возможностей, которые предоставляет операционная система для создания собственных алгоритмов управления учетными записями пользователей и получения различного рода информации о пользователях.

* * *

В этой главе были затронуты основные аспекты управления пользователями в операционной системе UNIX. В последующих главах будут обсуждаться другие вопросы, касающиеся работы пользователей в UNIX, в частности, управление файлами и процессами, которые тесно связаны с рассмотренным материалом.

Глава 4

Командный интерпретатор shell

При работе в операционной системе UNIX любой пользователь, независимо от того, какие задачи решает, сталкивается с необходимостью выполнения команд, запускаемых из окна консоли. Мы довольно часто использовали команды UNIX в предыдущих главах и будем это делать в дальнейшем. С их помощью можно выполнять большинство действий по настройке, диагностированию и управлению системой. Команды операционной системы являются "кирпичиками", из которых строятся любые, часто очень сложные и изощренные программы.

Команды UNIX не выполняются сами по себе, а только в контексте определенной командной оболочки, которую называют интерпретатором команд. Интерпретатор команд проверяет и анализирует введенные команды и их аргументы, анализирует их синтаксис, корректность введенных ключей и т. д. После успешного завершения проверки интерпретатор запускает соответствующую программу, т. е. создает в UNIX процесс и передает ему управление.

Таким командным интерпретатором в UNIX является shell. Помимо исполнения команд shell выполняет и другие не менее важные операции: конвейеризацию команд, переназначение ввода/вывода, генерацию имен файлов, контроль среды окружения. Значение интерпретатора для пользователей UNIX, особенно для тех, кто занимается настройкой и администрированием системы, трудно переоценить. Сложные задачи, требующие выполнения определенной последовательности команд, легко реализуются с помощью так называемых командных файлов (скриптов) или, по-другому, командных сценариев, написанных с использованием интерпретатора shell. Хочу отметить, что в дальнейшем выражения "командный файл", "командный скрипт" и "скрипт" будут использоваться как синонимы и обозначать исполняемый файл, состоящий из команд в виде текстовых строк.

В настоящее время наиболее часто используются четыре основные разновидности shell: Bourne shell (sh), C shell (csh), Korn shell (ksh) и Bourne Again shell (bash). Bourne shell является оригинальным UNIX shell и присутствует во всех без исключения дистрибутивах операционной системы UNIX.

Интерпретатор C shell разработан в Калифорнийском университете (г. Беркли). Особенностью этого интерпретатора является возможность интерактивной обработки shell-окружения.

Korn shell разработан Дэвидом Корном и включает целый ряд дополнительных возможностей по сравнению с Bourne shell.

Bourne Again shell разработан Фондом свободно распространяемых программных продуктов (Free Software Foundation) и аккумулирует в себе возможности оболочек C и Korn, что обусловило его широкую популярность среди опытных пользователей и системных администраторов. Он является наиболее продвинутым интерпретатором по сравнению с остальными и служит очень мощным инструментом программирования задач системного администрирования. В этой главе мы будем подробно рассматривать именно возможности bash, хотя большинство примеров задач без каких-либо изменений будут работать и в других shell.

В дальнейшем мы будем использовать наряду с термином "интерпретатор" другое определение — "командная оболочка". Оба эти выражения являются синонимами и употребляются в одинаковом контексте.

Одна и та же операционная система позволяет работать сразу с несколькими командными оболочками и переходить от одной командной оболочки к другой. Возможна и одновременная работа пользователя в нескольких экранах со своими командными оболочками.

Командный интерпретатор shell в практическом аспекте представляет собой язык программирования очень высокого уровня. Работу с командной оболочкой можно начинать после входа пользователя в систему. Оболочка может отображать приглашение (prompt) к вводу команды в виде одного из символов `$`, `#`, `>` или др., после которого обычно следует пробел. Команды пользователя вводятся сразу же после этого пробела. Рассмотрим наиболее важные особенности shell. В качестве командной оболочки выберем bash и на практических примерах проанализируем особенности его применения.

Исходные тексты командных файлов, представленных в данной главе, можно легко применить на практике, причем подавляющее большинство командных файлов без каких-либо изменений будет работать в любой из версий UNIX. При разработке примеров я старался избегать применения специфичных для

какой-либо операционной системы команд. Все примеры разработаны для наиболее популярной версии интерпретатора bash.

Итак, приступим к более подробному изучению командной оболочки shell.

4.1. Синтаксис shell

Анализ синтаксиса командного интерпретатора shell начнем с применения символов, имеющих специальное значение и используемых для организации выполнения нескольких команд из командной строки. Одним из таких разделителей является точка с запятой (`;`). Следующий пример демонстрирует использование точки с запятой:

```
# who;ls -l
root     :0           Jan 17 09:32
root     pts/0        Jan 17 09:57 (:0.0)
total 52
-rw-r--r--    1 root     root   0     Jan  6 14:41 a.out
drwxr-xr-x    8 root     root   4096  Jan 16 23:43 developer
-rw-r--r--    1 root     root   20633 Dec  1 12:50 install.log
-rw-r--r--    1 root     root   3841  Dec  1 12:50 install.log.syslog
-rw-r--r--    1 root     root   479   Jan 17 09:40 list_dir
```

В этом примере последовательно выполняются две команды — `who` и `ls -l`, первая из которых отображает на экране пользователей, работающих в системе, а вторая — список файлов и каталогов.

Последовательность символов `&&` (двойной амперсанд) читает код завершения первой команды, и если он равен 0 (успешное завершение), то будет выполнена вторая команда, например,

```
# test -d ./developer && echo developer is a directory
developer is a directory
```

При этом сообщение выводится только в том случае, если объект файловой системы /developer является каталогом, что действительно так.

Если выполнить команду

```
# test -f ./developer && echo developer is a directory
```

то никакого сообщения выводиться не будет, поскольку результатом выполнения команды `test` является число, отличное от нуля (/developer не является файлом).

Следующая командная строка

```
# test -e a.out && echo a.out exists
a.out exists
```

выполняет проверку на существование файла a.out в текущем каталоге и выводит соответствующее сообщение.

Еще пример. Команда

```
# test -r list_dir && cat list_dir
```

выводит на экран содержимое файла list_dir, если этот файл имеет установленный атрибут для чтения.

Противоположный результат дает применение оператора `||`. Если две команды разделены этим оператором и первая команда завершается с ненулевым результатом (т. е. неудачно), то выполняется вторая команда. Успешное выполнение первой команды (нулевой код завершения) приведет к тому, что вторая выполняться не будет. Например, команда:

```
# test -f /home/developer || echo developer is a directory
```

будет иметь результатом строку на экране

```
developer is a directory
```

Аналогично команда

```
# test -x list_dir || cat list_dir
```

отображает содержимое файла list_dir, поскольку для этого файла не установлен атрибут исполнения, и результат выполнения команды `test -x` будет ненулевым.

С помощью операторов `&&` и `||` можно объединять более двух команд, создавая при этом замысловатые алгоритмы обработки. Например, следующая командная строка

```
# test -e list_dir && test -r list_dir && cat list_dir
```

перед выводом содержимого файла list_dir на экран проверяет существование файла (`test -e`) и доступ к нему по чтению (`test -r`), и только при выполнении этих двух условий выполняется команда `cat`.

Можно выполнить как альтернативную и такую команду:

```
# test -e list_dir && test -x list_dir || cat list_dir
```

В этом случае содержимое файла list_dir также будет выведено на экран, поскольку команда `test -e` завершается успешно, вызывая на выполнение команду `test -x`, которая завершается с ненулевым кодом, что, в свою очередь, позволяет выполниться команде `cat`.

Следующая командная строка не позволит вывести на экран содержимое файла list_dir:

```
# test -e list_dir && test -x list_dir && cat list_dir
```

поскольку команда `test -x` завершится неудачно.

Следует помнить, что результат выполнения таких цепочечных конструкций определяется результатом выполнения каждой из команд.

Если команда должна выполняться как фоновый процесс, то в ее конце нужно указать символ амперсанда (`&`), например, как в этом случае:

```
# ls -l&
[1] 4840
total 52
-rw-r--r--    1 root     root  1544 Dec  1  12:54 anaconda-ks.cfg
. . .
-rw-r--r--    1 root     root  479  Jan  17 09:40 list_dir
#
```

Заметьте, что при выполнении команды в фоновом режиме на экран выводится номер процесса, соответствующий выполняемой команде. Система, запустив этот фоновый процесс, вновь выходит на диалог с пользователем.

В фоновом режиме можно заставить выполняться и несколько команд, разделенных точкой с запятой:

```
# who;ls -l;pwd&
root     :0           Jan 17 09:32
root     pts/0        Jan 17 09:57 (:0.0)
[1]+  Done
total 56
-rw-r--r--    1 root     root   1544 Dec  1 12:54 anaconda-ks.cfg
. . .
-rw-r--r--    1 root     root   479 Jan 17 09:40 list_dir
[1] 4843
# /root
```

Здесь на экран последовательно выводятся информация о пользователях, зарегистрированных в системе, содержимое текущего каталога и сам текущий каталог (/root).

Запустить на выполнение в фоновом режиме можно и цепочку из нескольких команд, например,

```
# test -e list_dir && test -x list_dir || cat list_dir&
```

4.2. Ввод/вывод

Одним из мощных средств командного интерпретатора shell является операция переназначения ввода/вывода. Вначале проанализируем, как осуществляется ввод/вывод в командной оболочке.

Стандартный ввод обозначается в операционной системе UNIX как stdin (standard input) и осуществляет операцию ввода данных с клавиатуры терминала. Стандартный вывод stdout (standard output) выполняет вывод данных на экран терминала. Кроме того, диагностические сообщения и сообщения об ошибках направляются в стандартное устройство ошибок, обозначаемое как stderr (standard error).

Операционная система UNIX обладает механизмами, позволяющими перенаправить результаты работы любой команды, предназначенные для стандартного ввода/вывода, на другое устройство или в файл. Вначале остановимся на переназначении вывода.

Поскольку любые устройства UNIX являются файлами, то принято говорить о переназначении вывода в файл, для чего служит оператор >. Если, например, нужно записать содержимое текущего каталога в файл с именем list_dir, то следует выполнить команду

```
# ls -l > list_dir
```

После выполнения этой команды файл list_dir будет содержать список файлов и каталогов. Чтобы убедиться в этом, наберите команду

```
# cat list_dir
```

Очень часто оператор переназначения вывода вместе с командой `echo` используется для записи текстовых строк в файл, например:

```
# echo This string will be written in file > textfile
```

К операторам переназначения вывода относится и >>. Он используется для записи данных в уже существующий файл, при этом информация записывается в конец файла. При помощи этого оператора можно записать в файл textfile из предыдущего примера еще одну строку:

```
# echo This will be added to the file >> textfile
```

В результате после выполнения этих двух команд файл textfile будет содержать строки

```
This string will be written in file
This will be added to the file
```

Должен заметить, что оператор переназначения >> можно использовать как замену >.

Если необходимо переназначить стандартное устройство ошибок, используйте выражение `2>`. Продемонстрирую это на примере выполнения команды `ls`, задав неправильный параметр `-y`, а результат выполнения команды перенаправим в файл ls_err вместо стандартного устройства ошибок stderr:

```
# ls -y 2> ls_err
```

После выполнения команды файл ls_err может содержать примерно такие записи:

```
# cat ls_err
ls: invalid option -- y
Try `ls --help` for more information.
```

При переназначении стандартного вывода следует помнить, что если файл с подобным именем существует, его предыдущее содержимое теряется. Если нужно сохранить содержимое файла, лучше воспользоваться оператором `>>` для добавления данных в конец файла.

Подобно тому, как выходные данные можно перенаправить в файл, входные данные можно получить из файла при помощи оператора `<` переназначения ввода. Следует учитывать то, что ввод из файла можно осуществить только для команд, допускающих получение данных из стандартного ввода.

Например, подсчитать количество строк файла с именем userfile можно с помощью команды

```
# wc — l  <  userfile
```

Ввод и вывод одной и той же команды могут быть переназначены одновременно, как в этом примере:

```
# echo This is a line written in file > input_output
# cat < input_output > ioresult
```

Команда `echo` записывает текстовую строку в файл input_output, содержимое которого является источником данных для команды `cat`. В свою очередь, вывод команды `cat` перенаправляется в файл ioresult. После выполнения команд файл ioresult содержит строку

```
This is a line written in file
```

Очень мощными средствами командного интерпретатора shell являются фильтры и конвейеры. Фильтрами называются программы или команды, которые выполняют чтение со стандартного ввода и записывают результат в стандартный вывод. Многие команды операционной системы UNIX реализованы в виде фильтров. Фильтры можно объединять, используя механизм конвейеризации. Средство, объединяющее стандартный вывод одной команды

со стандартным вводом другой, называется конвейером (pipe) и обозначается вертикальной чертой (|).

Командная строка, состоящая из нескольких команд, связанных конвейерами, может выглядеть так:

команда1 | *команда2* | *команда3*...

Самая первая команда конвейера — *команда1* — создает поток выходных данных, являющихся входными данными для команды *команда2*. В свою очередь, вывод команды *команда2* является вводом для команды *команда3* и т. д.

Преимущество использования конвейеризации можно показать, сравнив два примера, выполняющих одно и то же действие, например, определение количества файлов, содержащихся в текущем каталоге.

Такую операцию можно выполнить при помощи двух команд:

```
# ls -l  > tmp
# wc -l < tmp
```

Здесь используются операторы переназначения ввода/вывода, а для хранения промежуточных результатов создается файл tmp. Более изящно эту же операцию можно реализовать при помощи конвейера:

```
# ls -l | wc -l
```

Комбинируя операторы переназначения ввода/вывода и конвейер, можно реализовать довольно сложные командные сценарии. Например, выполнение команд

```
# echo The number of directory entries is > list_dir
# ls | wc -l  >> list_dir
```

позволяет записать в файл list_dir количество файлов текущего каталога, предварив их текстовой строкой. Конвейеризация, как мы увидим далее, очень широко используется при написании командных файлов.

4.3. Командные файлы

Наиболее ценным свойством командного интерпретатора shell является то, что последовательность команд можно записать в текстовый файл и затем использовать для выполнения последовательности операций. Напомню, что такой текстовый файл называют командным файлом, командным сценарием или командным скриптом.

Рассмотрим примеры создания и запуска командных файлов. С помощью текстового редактора (можно использовать `vi` или другой) запишем в файл (назовем его dl) строки:

```
pwd
ls
echo This is the end of the shell script
```

Чтобы выполнить команды, записанные в файле dl, наберем строку

```
# sh dl
```

Если текущим каталогом является, например, /home/user1, то результат может быть примерно таким:

```
# sh dl
/home/user1
file1
file2
file3
This is the end of the shell script
```

Еще один способ выполнить команды, находящиеся в текстовом файле, — сделать его исполняемым (командным) при помощи команды `chmod`:

```
# chmod +x имя_файла
```

Здесь *имя_файла* — имя текстового файла, подлежащего выполнению. Данная команда устанавливает для файла атрибут исполнения (в символьной форме `+x`). При указании имени файла нужно помнить, что UNIX различает строчные и прописные литеры.

При написании командных файлов следует учитывать несколько важных моментов. Нежелательно начинать командный файл символом `#`, хотя вполне естественно, когда исходный текст предваряется комментарием. Это связано с тем, что такой командный файл в оболочке C shell (csh) будет интерпретирован как выполняемый, в результате будет активизирован интерпретатор csh. Лучше всего, если командный файл начинается с пустой строки или с пустого оператора `:`.

Ни в коем случае не следует присваивать командному файлу имя, совпадающее с именем одной из системных команд, таких, например, как `ls`, `rm` или `mv`, поскольку вместо командного файла UNIX выполнит системную команду. Это объясняется тем, что система ищет выполняемую команду в каталогах, определяемых системной переменной `PATH`, и только потом в каталоге пользователя.

4.4. Переменные

Перейдем к описанию синтаксиса командного интерпретатора и начнем с представления переменных. Как и любой язык программирования высокого уровня, командный интерпретатор допускает использование переменных. Переменные, так же как и командные файлы, являются базовыми объектами командной оболочки. Выделяют несколько типов переменных: позиционные параметры, ключевые и специальные параметры.

Позиционные параметры (positional parameters) позволяют задавать аргументы командной строки для выполняемого файла. Позиционный параметр обозначается как число, перед которым расположен знак доллара, например, `$1`, `$2`, `$3` и т. д. При выполнении командного файла параметр `$1` заменяется аргументом, стоящим первым после имени командного файла, параметр `$2` заменяется вторым аргументом и т. д. При этом в командном файле можно использовать до девяти позиционных параметров.

Проанализируем выполнение следующего примера. Создадим командный файл, записав в него такие строки:

```
echo The first parameter is: $1
echo The second parameter is: $2
echo The third parameter is: $3
echo The fourth parameter is: $4
echo The fifth parameter is: $5
```

Сохраним файл под именем shell.prog и сделаем его исполняемым:

```
# chmod +x shell.prog
```

При запуске командного файла с параметрами `one`, `two`, `three`, `four`, `five` получим такой результат:

```
# shell.prog one two three four five
The first parameter is: one
The second parameter is: two
The third parameter is: three
The fourth parameter is: four
The fifth parameter is: five
```

Позиционные параметры позволяют задавать параметры командной строки в командных файлах, включая применение в конвейере команд. Вот некоторые примеры использования позиционных параметров.

Запишем следующую строку в файл echo_mail:

```
echo $1 | mail $2
```

Здесь первый параметр `$1` позволяет задавать текст почтового сообщения, а второй параметр — имя почтового клиента. Установим атрибут исполнения с помощью команды `chmod +x echo_mail` и выполним командный файл:

```
# ./echo_mail "TEST MESSAGE FOR root" root
```

Можно проверить работу этого командного файла, прочитав сообщение:

```
# mail
Mail version 8.1 6/6/93.  help.
"/var/spool/mail/yury": 1 message 1 new
>N  1 yurylinx  Sun Jul 11 20:56  15/656
& 1
Message 1:
From yury@localhost.localdomain  Sun Jul 11 20:56:36 2004
Date: Sun, 11 Jul 2005 20:56:36 +0300
From: Yury <yury@localhost.localdomain>
To: root@localhost.localdomain

TEST MESSAGE FOR root
&
```

Следующий пример показывает вывод строки или строк содержимого каталога, удовлетворяющих значению параметра `$1`. Запишем в командный файл (назовем его demo_pos_param) строку

```
ls -l | grep $1
```

Выполнение этого командного файла с параметром `mail*` в моей операционной системе дает, например, такой результат:

```
 # ./demo_pos_param mail*
-rw-rw-r--    1 yury     yury   19 Jul  8 22:58 mail_note
```

Хочу сделать замечание относительно применения позиционных параметров. Командная строка shell может содержать до 128 аргументов, однако командный скрипт способен ссылаться одновременно только на девять параметров.

Кроме позиционных параметров в командных файлах применяют и другой тип переменных, называемых ключевыми параметрами (keyword parameters). В дальнейшем мы будем использовать термин "переменная", подразумевая при этом именно ключевой параметр.

При задании ключевого параметра имя переменной должно начинаться с буквы или с символа подчеркивания, например:

```
_myfile
```

```
output_File_1
Input_Value
```

Значения переменным можно присвоить с помощью оператора присваивания =:

```
Myvar=var1
Var1=1
```

Для переменной, содержащей строку, значение присваивается так:

```
Message="Users will be off during 5 min."
```

Обратите внимание на то, что строка заключается в кавычки.

Имена переменных наподобие

```
1rs
my string
.var1
```

использовать нельзя. Первое имя (`1rs`) начинается с цифры, второе (`my string`) — содержит пробел, третье (`.var1`) — содержит недопустимый символ — точку.

В отличие от некоторых других языков программирования, например, C, переменные shell не связаны с определенным типом данных, и любое значение, которое присваивается таким переменным, интерпретируется командной оболочкой как строка символов.

Присвоив переменным определенные значения, можно использовать их в программе. Для доступа к переменной нужно непосредственно перед ее именем установить знак доллара (`$`). Следующий пример демонстрирует вывод содержимого переменной `var` на экран:

```
# var=1
# echo $var
1
#
```

Отображение строки на экран можно получить так:

```
# message="Hello from Shell!"
# echo $message
Hello from Shell!
#
```

Везде, где встречается символ `$`, предшествующий имени переменной, он заменяется ее значением, например:

```
# myvar=10
```

```
# echo The value of myvar is $myvar
The value of myvar is 10
#
```

Если имя переменной заключить в фигурные скобки, то отображаемое на дисплее значение будет содержать символы (если есть), находящиеся за фигурными скобками:

```
# myvar="String 1"
# echo ${myvar}1
String 11
#
```

Обратите внимание на то, что при выполнении операции присваивания переменная и присваиваемое ей значение должны быть записаны без пробелов относительно оператора =.

В качестве значения переменная может принимать и результат выполнения команды, как в следующем примере:

```
# DATE=`date`
```

Такая форма записи команды позволяет вначале выполнить команду `date`, после чего результат ее выполнения присвоить переменной `DATE`. При этом результат выполнения команды `date` не выводится на стандартный вывод. Обратите особое внимание на то, что команда `date` заключена в обратные апострофы!

Во многих случаях требуется присваивать переменной определенные значения, используя клавиатуру консоли. В таких случаях используется команда `read`, которой довольно часто предшествует команда `echo`, отображающая на экране приглашение к вводу, например:

```
echo "Enter integer:"
read x
```

При выполнении этого фрагмента командного файла после вывода на экран сообщения

```
Enter integer:
```

интерпретатор shell будет ожидать ввода значения с клавиатуры. Одна команда `read` позволяет присвоить значения сразу нескольким переменным. Следует учитывать, что если параметров команды `read` больше, чем их введено, то оставшимся переменным присваивается пустая строка. Если количество введенных значений больше, чем параметров в команде `read`, то лишние игнорируются.

С помощью команды `read` можно считывать значения нескольких переменных, например, как в этом командном файле (назовем его read_demo):

```
echo Enter string:
read x y
echo You entered: $x + $y
```

Результат работы командного файла показан далее:

```
# ./read_demo
Enter string:
34 56
You entered: 34 + 56
```

К ключевым параметрам относятся и так называемые предопределенные переменные. Любой процесс имеет среду (environment), в которой он выполняется, представляющую собой в первом приближении набор некоторых переменных и констант, доступных данному процессу. Этот набор выделяется процессу операционной системой и может использоваться, например, для установки параметров по умолчанию для терминальных линий, ссылок на определенные каталоги, содержащие команды операционной системы, и т. д.

Можно просмотреть информацию о наиболее важных стандартных переменных с помощью команды `set`. К таким переменным можно отнести как переменные процесса, так и экспортируемые переменные, созданные другими процессами. Часто такие переменные называют переменными окружения.

Вид и содержание выводимой информации зависят от версии UNIX, в которой работает пользователь. Перечислим некоторые важнейшие системные переменные:

- `PATH` — указывает, в каких каталогах искать выполняемые файлы. Об этой переменной поговорим немного позже;
- `HOME` — это имя домашнего каталога, в котором пользователь оказывается после входа в систему;
- `MAIL` — имя файла, в который поступает электронная почта;
- `SHELL` — указывает оболочку, в которой работает пользователь. В данном случае используется расширенная версия (bash).

Одной из важнейших переменных является `PATH`. Она показывает путь, где нужно искать команду, и после входа пользователя в систему принимает определенное значение по умолчанию. Просмотреть это значение можно, задав команду

```
# echo $PATH
```

Переменная `PATH` содержит список всех каталогов, где нужно искать команду или исполняемый файл, и, кроме того, указывается последовательность пере-

бора путей поиска. При этом каталоги, указанные в PATH, просматриваются слева направо и отделяются друг от друга двоеточием.

Иногда требуется изменить путь доступа или добавить новый каталог для поиска. В таких случаях следует скорректировать переменную PATH, например:

```
PATH=/bin:/sbin:/usr/bin:/home/developer
```

В этом примере к списку путей поиска добавляется путь /home/developer. Эту же операцию можно выполнить и по-другому, присоединив новый каталог к концу существующего в PATH:

```
PATH=$PATH:/home/developer
```

Кроме рассмотренных, существуют еще два так называемых специальных параметра, обозначаемых как $# и $*.

Параметр $# указывает на общее количество параметров, с которыми вызывается командный файл. Использование $# покажем на примере. Сохраним строку

```
echo The number of arguments is: $#
```

в файле arg_num и выполним его:

```
# arg_num Test program
The number of arguments is: 2
#
```

Параметр $* содержит строку, состоящую из всех аргументов, с которыми вызывается скрипт. В следующем примере создадим командный файл с именем show_params, содержащий строку

```
echo The parameters for this command are: $*
```

Для проверки работоспособности выполним командный файл с параметрами "Hello. How are you?" Результат может быть таким:

```
# show_params Hello. How are you?
The parameters for this command are: Hello. How are you?
```

4.5. Метасимволы

Командный интерпретатор позволяет использовать специальные символы (метасимволы) для работы со строками, файлами и каталогами. Следующие метасимволы очень часто применяют в командных файлах:

- * — произвольная последовательность символов, причем может не содержать ни одного символа;
- ? — один произвольный символ;

- `[...]` — любой из символов в указанном диапазоне либо указанный в перечислении.

Вот примеры использования метасимволов. Команда

```
# ls -l  t*
```

отображает все файлы каталога, начинающиеся с `t`.

Команда

```
# ls -l *t*
```

отображает на консоли все файлы, содержащие `t`.

Команда

```
# ls -l t.?
```

выводит файлы текущего каталога с однобуквенными расширениями, например, tx.c и tx.o.

Команда

```
# ls -l [a-d]*
```

выводит файлы, которые начинаются с `a`, `b`, `c`, `d`. Тот же самый результат можно получить, если ввести команды

```
# cat [abcd]*
# cat [bdac]*
```

Символ `*` можно использовать и с другими командами. Например, команда

```
# echo *
```

отображает имена всех файлов текущего каталога на экране.

Кроме перечисленных метасимволов, командный интерпретатор shell обрабатывает символы, имеющие специальное назначение:

- двойные кавычки (`"`);
- апостроф (`'`);
- символ обратной наклонной черты (`\`);
- обратный апостроф (`` ` ``).

Предположим, что текущий каталог содержит файлы data1, prog1 и text1. Введем две команды `echo`, аргументы которых содержат `*`, и сравним результаты выполнения:

```
# echo *
data1 prog1 text1
# echo  "*"
*
#
```

В первом случае интерпретатор shell воспринимает * как список содержимого текущего каталога и отображает список файлов, а во втором двойные кавычки отменяют значение символа *. В этом и состоит смысл использования двойных кавычек — любой символ, имеющий специальное значение (например, *, ?, >, >>, ||), утрачивает свой особый статус. Исключениями являются символ $, обратный апостроф (`) и обратная наклонная черта (\), если она предшествует специальному символу.

Рассмотрим применение апострофа, выполнив команду

```
# echo '$text'
$text
#
```

Как видно из примера, если переменная заключена в апострофы, то ее значение не подставляется в качестве результата. Чтобы увидеть разницу между двойными кавычками, апострофами и их отсутствием, выполним следующие команды:

```
# echo *
data1 prog1 text1

# echo * "*" '*'
data1 prog1 text1 * *

# var1=test
echo $var1 "$var1" '$var1'
test test $var1
#
```

Обратная наклонная черта перед символом ликвидирует его специальное значение для shell, например,

```
# echo \$text
$text
# echo \\
\
#
```

Если некоторая команда будет заключена в обратные апострофы, командная оболочка обязательно исполнит ее и поместит результат на то место, где эта команда находится, например:

```
# users=`who | wc -l`
# echo $users users logged in.
3 users logged in.
#
```

4.6. Вычисления

Большинство командных интерпретаторов в общем случае не предназначено для выполнения сложных математических вычислений, хотя имеет встроенные средства для арифметических операций. Приятным исключением является командная оболочка bash, в которой весьма существенно расширены возможности по обработке математических величин.

Вначале рассмотрим стандартные возможности, характерные для всех shell. Напомню, что все переменные в shell воспринимаются как строки. Например, командный интерпретатор выполняет приведенные далее операции следующим образом:

```
# n=0
# n=$n+1
# echo $n
0+1
#
```

Для обработки целых чисел можно воспользоваться командой `expr`. Проиллюстрируем некоторые возможности этой команды на примере. Следующий командный файл выполняет вывод разности двух целых чисел на дисплей:

```
echo Enter two integers
read x y
echo the difference = `expr $x - $y`
```

Обратите внимание на запись выражения справа от команды `expr`. Для получения правильного результата необходим пробел между переменными и знаком операции.

Следующий командный файл для вычисления суммы двух целых чисел (назовем его summa) работать не будет:

```
echo Enter two numbers:
read x y
echo First number: $x
echo Second number: $y
echo Summa = `expr $x+$y`
```

При выполнении этого командного файла на консоль выводится что-то наподобие

```
# ./summa
Enter two numbers:
4 7
```

```
First number: 4
Second number: 7
Summa = 4+7
```

Здесь вместо результата, равного 11, мы получили строку "4+7", поскольку интерпретатор не смог вычислить это выражение из-за неправильной записи выражения справа от `expr`.

Следующий командный файл также содержит ошибку:

```
echo Enter two numbers:
read x y
echo First number: $x
echo Second number: $y
z=11
echo Summa = `expr $x+$y - $z`
```

При выполнении этого командного файла интерпретатор сообщит об ошибке:

```
# ./summa
Enter two numbers:
4 5
First number: 4
Second number: 5
expr: non-numeric argument
Summa =
```

Предлагаю читателям проанализировать этот пример самостоятельно и найти ошибку.

С помощью команды `expr`, кроме суммы и разности, можно вычислять также произведение чисел или частное от их деления. Например, для нахождения произведения двух чисел, значения которых введены с клавиатуры, можно воспользоваться командным файлом, содержащим такие строки:

```
echo Enter two numbers:
read x y
echo First number: $x
echo Second number: $y
echo $x '*' $y = `expr $x '*' $y`
```

Работу скрипта (его можно назвать `multiply`) легко проверить:

```
# ./multiply
Enter two numbers:
3 26
```

```
First number: 3
Second number: 26
3 * 26 = 78
```

Обратите внимание на запись

```
`expr $x '*' $y`
```

Здесь для выполнения операции умножения используется символ `*`, взятый в апострофы. При отсутствии апострофов интерпретатор во время выполнения командного файла выдает ошибку наподобие:

```
# ./multiply
Enter two numbers:
3 26
First number: 3
Second number: 26
expr: syntax error
3 * 26 =
```

Если применяется операция умножения (`*`), то она обязательно должна быть заэкранирована, поскольку в командной оболочке этот символ воспринимается как символ подстановки.

Операцию деления двух целых чисел можно выполнить, если заменить символ `*` символом `/`, как, например, в скрипте `div`:

```
echo Enter two numbers:
read x y
echo First number: $x
echo Second number: $y
echo $x '/' $y = `expr $x '/' $y`
```

При выполнении деления двух целых чисел интерпретатор отбрасывает остаток, как, например, в этом случае:

```
# ./div
Enter two numbers:
39 4
First number: 39
Second number: 4
39 / 4 = 9
```

Комбинируя различные операции, можно выполнять и более сложные вычисления:

```
echo Enter two numbers:
read x y
```

```
echo First number: $x
echo Second number: $y
echo $x '*' $y + $x '/' $y = `expr $x '*' $y + $x '/' $y`
```

Далее представлен результат работы командного файла (он называется combo):

```
# ./combo
Enter two numbers:
7 4
First number: 7
Second number: 4
7 * 4 + 7 / 4 = 29
```

Автор предлагает читателям проанализировать результат самостоятельно.

Напомню, что использование комментария в виде символа # допустимо во всех командных оболочках, кроме csh. Между знаками операций и переменными обязательно должны находиться пробелы.

4.7. Общие переменные

Нередко требуется передавать значения переменных, определенных в одном командном файле, другому процессу. Напомню, что переменные являются локальными для процесса, в котором они определены, т. е. там, где им присвоены значения. Для того чтобы они были доступны другим процессам, переменные нужно передать этим процессам явным образом или экспортировать.

В командном интерпретаторе bash для этих целей существует команда export, имеющая синтаксис:

```
# export список_переменных
```

Здесь *список_переменных* представляет собой список переменных, разделенных пробелами, причем символ $ перед именами переменных не ставится. Любой процесс, запущенный после этой команды, получает доступ к этим переменным, но изменить их значений он не может.

Рассмотрим пример. Введем такую последовательность команд:

```
# cat > demo_export
echo $x
Ctrl-D
#
```

Командный файл demo_export содержит команду echo $x, отображающую значение переменной x на экране консоли. Если теперь выполнить этот файл,

то ничего на экране мы не увидим, поскольку команда `echo` ничего не знает о переменной `x`.

Теперь присвоим переменной `x` значение 111 и экспортируем ее:

```
# x=111
# export x
```

Еще раз выполним командный файл demo_export:

```
# ./demo_export
111
#
```

Сейчас все работает, как надо. Если изменить значение переменной `x` на другое, то новое значение будет доступно другим процессам, так что команду `export` повторно выполнять не нужно.

Устанавливаемые переменные окружения, так же как и другие переменные, можно экспортировать. Обычно такая возможность используется в процессе настройки программных продуктов. Например, для экспорта переменной окружения `PATH` нужно ввести команду

```
export PATH MAIL
```

4.8. Логические структуры

До сих пор мы рассматривали командные файлы, в которых команды выполнялись последовательно. Однако ни один язык высокого уровня, в том числе и shell, не может обходиться без программных конструкций, позволяющих изменять последовательность выполнения команд в зависимости от тех или иных условий. Такие программные конструкции очень часто называют логическими структурами. Благодаря им язык командного интерпретатора приобретает особую гибкость и мощь. Следует сказать, что почти все конструкции shell похожи на аналогичные им логические структуры, присутствующие в языках высокого уровня, таких, например, как C или Fortran.

Практически все логические структуры командного интерпретатора используют коды завершения команд для определения дальнейшей последовательности выполнения команд. Код завершения показывает, успешно или неудачно выполнилась команда. Все команды shell возвращают 0 в случае удачного завершения и значение, отличное от нуля, в случае неудачи.

Код завершения команды не отображается автоматически на экране — для этого необходимо задавать специальный параметр `$?`, как показано в примере:

```
# pwd
/root
```

```
# echo $?
0
# cat file1
cat: file1: No such file or directory
# echo $?
1
#
```

Здесь следует сделать важное замечание: почти все логические структуры используются только в командных файлах и не могут задаваться в командной строке. Исключением является команда `test`, которую можно использовать из командной строки.

4.8.1. Оператор цикла *for*

Рассмотрим более подробно логические структуры командного интерпретатора shell и начнем с оператора цикла `for`. В общем виде цикл `for` можно представить следующим образом:

```
for переменная in список_переменных
do
  команда_1
  команда_2
  . . .
  команда_n
done
```

Здесь жирным шрифтом выделены ключевые слова, используемые в цикле `for`. Цикл `for` выполняет последовательно команды, находящиеся между ключевыми словами `do` и `done`. При этом очередное выбираемое значение из списка *список_переменных* присваивается переменной *переменная*, имя которой указано слева от ключевого слова `in`. Список значений может задаваться как символьными константами, так и значением некоторой переменной, содержащей последовательность значений, отделенных разделителями, обычно пробелами.

Следующий командный файл (назовем его demo_for_1) демонстрирует базовые свойства цикла `for`:

```
for x in  9 8 7
do
 echo "$x"
done
echo
```

Результатом выполнения файла demo_for_1 является последовательность чисел:

```
# ./demo_for
9
8
7
#
```

Командный файл demo_for_2, исходный текст которого показан далее, выводит на экран дисплея содержимое всех объектов текущего каталога:

```
for x in  *
do
 echo "============= $x =============="
 cat "$x"
 echo
done
echo
```

Здесь символ `*` в операторе `for` указывает на необходимость просмотра всех элементов текущего каталога. Если в каталоге находятся, например, такие элементы, как показано далее

```
# ls -l
total 7
-rwxrwxrwx    1 YURY     Отсутств       45 Jan 23 14:32 demo_for_1
-rwxrwxrwx    1 YURY     Отсутств       88 Jan 23 14:38 demo_for_2
-rw-r--r--    1 YURY     Отсутств       27 Jan 23 14:43 text1
-rw-r--r--    1 YURY     Отсутств       27 Jan 23 14:43 text2
```

то результат работы командного файла demo_for_2 может выглядеть так:

```
============= demo_for_1 ==============
for x in  9 8 7
do
 echo "$x"
done
echo

============= demo_for_2 ==============
for x in  *
do
 echo "============= $x =============="
```

```
 cat "$x"
 echo
done
echo

============ text1 ==============
this is a content of text1

============ text2 ==============
this is a content of text2
```

Следующий пример показывает, как можно переместить файлы из каталогов, указанных в списке переменных, в другой каталог. Исходный текст командного файла показан далее:

```
echo enter the directory path
read path
test -e $path || mkdir $path
for file in dir1 dir2 dir3
 do
   mv $file $path/$file
done
```

Здесь команда `test -e $path || mkdir $path` создает каталог с именем, заданным переменной `path`, если таковой отсутствует.

Командный файл, показанный далее (назовем его all_in_one), выполняет слияние содержимого нескольких файлов, начинающихся с литеры "t", в один файл с именем newt. Здесь также используется цикл `for`:

```
for x in t*
 do
  cat $x >> newt
 done
```

Закончим демонстрацию возможностей оператора `for` командным файлом, выполняющим удаление файлов нулевого размера из текущего каталога. Вот исходный текст командного файла (назовем его remove_empty):

```
echo Removing empty files
for file in *
 do
  test -d $file && echo "File $file is directory. Skipped" && continue
  test -s $file && continue
```

```
  echo "Removing $file..." && unlink $file
 done
echo
```

Остановимся более подробно на анализе исходного текста этого командного файла. Команда

```
test -d $file && echo "File $file is directory. Skipped" && continue
```

выполняет два действия:

- проверяет, является ли обрабатываемый элемент каталогом;
- если это каталог, то оставшиеся операторы цикла пропускаются (оператор `continue`), и анализируется следующий элемент каталога, иначе выполняются следующие операторы цикла `for`.

Команда

```
test -s $file && continue
```

определяет, равен ли размер файла нулю, и если это так, то выполняется следующая команда цикла, при помощи которой файл удаляется. Непосредственное удаление файла выполняется командой `unlink`.

Предположим, что перед выполнением командного файла remove_empty содержимое текущего каталога было следующим:

```
# ls -l
total 20
drwxr-xr-x    5 root     root   4096  Jan 23 16:04 backup
-rwxr-xr-x    1 root     root   194   Jan 23 17:59 demo_2
-rw-r--r--    1 root     root   0     Jan 23 18:07 empty1
-rw-r--r--    1 root     root   0     Jan 23 18:07 empty2
-rw-r--r--    1 root     root   1504  Jan 23 17:57 Using_operator_FOR
drwxr-xr-x    2 root     root   4096  Jan 23 17:45 usr1
```

Во время выполнения командного файла будет отображаться такая информация:

```
# ./remove_empty
Removing empty files
File backup is directory. Skip
Removing empty1...
Removing empty2...
File usr1 is directory. Skip
```

В результате выполнения командного файла файлы empty1 и empty2 будут удалены.

4.8.2. Оператор условия *if*

Следующая логическая структура высокого уровня, которую мы рассмотрим, и которая используется очень часто, — оператор условия `if`.

Ни один язык высокого уровня не обходится без операторов условия, и командный интерпретатор shell также не является исключением. В общем виде оператор `if` можно представить так:

```
if условие
then
   команда_1
   команда_2
   . . .
   команда_n
fi
```

Если *условие* истинно, то выполняются команды, находящиеся между `then` и `if`, а если ложно — будут пропущены. Условие представляет собой определенное выражение, заключенное в квадратные скобки, например,

```
[ $I -gt 7 -a $I -le 100 ]
```

Это выражение будет истинно, если переменная `I` больше 7 и меньше или равна 100.

Оператор `if` нельзя выполнить из командной строки, поэтому его используют только при разработке командных файлов. Рассмотрим несколько примеров применения оператора `if`.

Для поиска слова в текстовом файле можно создать командный файл, исходный текст которого представлен далее:

```
echo Enter the word and filename
read word file
if grep $word $file
  then
    echo $word found in $file
fi
```

Выполняющаяся программа ожидает ввода с консоли искомого слова и имени файла, после чего анализирует содержимое файла. Если слово найдено, то выводится сообщение на экран дисплея.

Имеется еще одна форма структуры `if`, в которой используется оператор `else`:

```
if условие
then
```

```
   команда_1
   команда_2
   . . .
else
   команда_3
   команда_4
   . . .
fi
```

Модифицируем предыдущий пример таким образом, чтобы в нем использовался оператор `if...else`:

```
echo Enter the word and filename
read word file
if grep $word $file >/dev/null
  then
    echo $word found in $file
  else
    echo $word is NOT found in $file
fi
```

В этом случае при обнаружении слова в файле выводится сообщение

```
echo $word found in $file
```

а при отсутствии этого слова — сообщение

```
echo $word is NOT found in $file
```

Замечу, что в данном командном файле используется устройство /dev/null, являющееся приемником ненужной или нежелательной информации в результате выполнения команд. Переназначение вывода в это устройство применяется в тех случаях, когда нужно "удалить мусор".

В следующем примере используется оператор `[...]` и обратные апострофы. Этот командный файл, как и предыдущий, ищет слово в текстовом файле с именем test, но при запуске требует ввода параметров из командной строки:

```
if [ $# -ne 1 ]
  then
    echo "Usage: example_if_grep_1 string"
  else
    output=`grep "$1" test`
    if [ -z "$output" ]
      then
```

```
        echo "Cannot find $1 in test1.txt"
    else
        echo "$output"
    fi
fi
```

Команда `if` в первой строке анализирует количество параметров командной строки. В случае их отсутствия или неправильного количества на экране отображается соответствующее сообщение и происходит выход из программы. Обратите внимание на строки

```
output=`grep "$1" test1.txt`
if [ -z "$output" ]
```

В первой из них результат выполнения команды `grep` присваивается переменной `output`, после чего проверяется, не является ли полученное значение пустой строкой. Остальная часть командного файла, думаю, понятна и в дополнительных объяснениях не нуждается.

Нередко в командных файлах применяется команда `test`. С ней мы уже сталкивались при анализе предыдущих примеров, сейчас же рассмотрим ее более подробно. Данная команда проверяет истинность определенных условий и полезна при организации ветвлений в программе. Наиболее часто команда `test` применяется вместе с оператором `if`.

Рассмотрим некоторые опции команды `test`:

- `-r` *файл* — условие истинно, если файл существует и доступен для чтения;
- `-w` *файл* — истинно, если файл существует и доступен для записи;
- `-x` *файл* — истинно, если файл существует и доступен для выполнения;
- `-s` *файл* — истинно, если файл существует и его размер больше нуля;
- *переменная_1* `-eq` *переменная_2* — истинно, если *переменная_1* равна *переменная_2*;
- *переменная_1* `-ne` *переменная_2* — истинно, если *переменная_1* не равна *переменная_2*.

Рассмотрим пример. Отобразим на экране дисплея содержимое текстового файла, имя которого будет задано в командной строке. С этим заданием справится несложный командный файл, исходный текст которого представлен далее:

```
if test -f $1
   then cat $1
fi
```

Следующий пример более сложен. Командный файл, исходный текст которого представлен далее, позволяет переместить все исполняемые файлы из текущего каталога программы в каталог, указанный аргументом командной строки. Здесь используются логические конструкции `for`, `if` и `test`:

```
for file in *
do
  if ! test -d $1
   then
     mkdir $1
  fi
  test -x $file && mv $file $1/$file
done
```

Предположим, что исходный текст сохранен в командном файле mvex. Тогда для перемещения исполняемых файлов в каталог backupex нужно ввести команду:

```
# ./mvex backupex
```

Если каталог, куда перемещаются файлы, не существует, то он создается. Вначале проверяется существование каталога (команда `if ! test -d $1`). Если каталог, указанный единственным аргументом программы, не существует, то он создается (команда `mkdir $1`).

В качестве дополнительного условия можно потребовать, чтобы размер перемещаемого файла был отличен от нуля. В этом случае предпоследняя строка предыдущего командного файла изменится (выделена жирным шрифтом):

```
for file in *
 do
  if ! test -d $1
   then
    mkdir $1
  fi
  test -x $file -a -s $file && mv $file $1/$file
 done
```

4.8.3. Операторы цикла *while* и *until*

Перейдем к следующему типу логических структур, известных как операторы цикла `while` и `until`. Начнем с оператора `while`. Этот оператор повторяет заданную группу команд, если условие выполнения соответствует истине.

Условие проверяется перед выполнением самой первой команды из списка команд, поэтому возможна ситуация, когда список не будет выполнен ни разу. Оператор `while` можно представить в следующем формате:

```
while условие
do
  команда_1
  команда_2
  . . .
  команда_n
done
```

Рассмотрим пример использования оператора `while`. Предположим, что необходимо записать символьные данные в файл с именем textfile. Для окончания записи данных нужно нажать комбинацию клавиш <Ctrl>+<D>, после чего содержимое файла будет отображено на дисплее. Исходный текст командного файла показан далее:

```
echo Please enter one or more words and press enter
echo Please end the list of words by pressing Control-D
while read x
do
 echo $x >> textfile
done
echo textfile contains the following words:
cat textfile
```

Что же касается оператора `until`, то он представляет собой разновидность оператора `while`. Основное отличие состоит в том, что в `while` цикл выполняется, пока условие является истинным, в то время как в `until` повторение цикла осуществляется до тех пор, пока условие ложно.

Вот пример применения оператора `until`. Командный файл, содержащий представленный далее исходный текст, каждые 60 секунд проверяет, есть ли на диске файл newf, и заканчивает работу, если он создан:

```
until test -r newf
do
  sleep 60
  echo Waiting until file is created...
done
echo File created
```

Следующий пример командного файла может помочь пользователю очистить каталог от файлов, имеющих нулевой размер:

```
for x in *
do until test -s "$x"
  do
    rm $x
    echo File $x is deleted
    break
  done
done
```

Здесь оператор `for` обеспечивает проход по всем файлам каталога, а команда `test` с опцией `-s` проверяет существование файла, а также его размера. Если файл имеет нулевой размер, команда `test -s` устанавливает код завершения, равный 1, что приводит к выполнению команды удаления файла `rm $x`. Далее цикл повторяется, пока не будут проверены все файлы каталога.

4.8.4. Оператор выбора *case*

Еще один оператор, используемый при создании командных файлов, — оператор `case`. Он удобен для организации ветвления программы и работает по принципу совпадения шаблонов. Вообще, оператор `case` можно заменить группой операторов `if...else`, однако во многих случаях использование `case` удобнее.

Оператор `case` имеет следующий формат:

```
case переменная
in
    шаблон_1) команда_11
              команда_12
              . . .
              команда_1N;;
    шаблон_2) команда_21
              команда_22
              . . .
              команда_2N;;
    шаблон_N) команда_N1
              команда_N2
              . . .
              команда_NN;;
esac
```

Здесь *переменная* сравнивается со всеми образцами, причем первое же совпадение приводит к выполнению соответствующей группы команд из списка. Образец *шаблон_N* может включать обычные метасимволы командной оболочки shell.

Рассмотрим пример командного файла, исходный текст которого показан далее:

```
echo Enter:
echo 1 - show current directory
echo 2 - show content
while read i
 do
 case $i in
   1)
      pwd
      continue;;
   2)
      ls -l
      continue;;
   *) break;;
 esac
done
```

Данный командный файл позволяет, в зависимости от выбранной опции, либо просмотреть текущий каталог, либо содержимое этого каталога. Здесь нам встретились две команды — `break` и `continue`. Команда `break` выполняет безусловный выход из тела цикла, а `continue` досрочно завершает текущую итерацию цикла `while` или `for` и переходит к следующей.

Следующий пример довольно сложен и требует от читателя повышенного внимания. Это исходный текст командного файла, предназначенного для выполнения операции резервного копирования и восстановления файлов. Здесь в качестве исходного каталога выбран $HOME/developer, а в качестве каталога, где создается копия, — backup.

В процессе архивирования выполняется копирование всего дерева каталогов, начиная с текущего:

```
echo "Select choice:"
echo "1 - backup files"
echo "2 - restore files"
echo "q - exit"
while read x
```

```
do
 case $x in
  1)cd $HOME
    rm -r backup
    cd developer
    find . -print -depth|cpio -pdm $HOME/backup
    continue;;
  2)cd $HOME
    test -d backup && cd backup && find . -print -depth|cpio -pdm
    $HOME/developer
    continue;;
  q)break;;
 esac
done
echo Done.
```

Исходный текст данного командного файла можно использовать в качестве шаблона для разработки собственной программы резервного копирования.

4.9. Потоковый редактор *sed*

Командная оболочка shell включает целый ряд дополнительных возможностей, позволяющих разработать командные файлы, выполняющие самые сложные задачи. Одна из таких возможностей предоставляется однопроходовым потоковым редактором `sed`, который наиболее часто применяется при разработке командных файлов.

Синтаксис команды в общем виде можно представить так:

```
sed команда_ed файл
```

Здесь *команда_ed* — команда, указывающая способ манипуляции с содержимым файла.

Указанная команда применяется ко всему файлу, если не определены конкретные строки для редактирования. При этом результат выполнения `sed` может быть перенаправлен на стандартный вывод или в другой файл. Мы не будем рассматривать все возможности редактора `sed`, тем более что эта тема хорошо описана в документации по UNIX.

Остановлюсь лишь на нескольких практических аспектах использования `sed`. С помощью этого потокового редактора можно вставлять, удалять и добавлять строки в текстовый файл, а также заменять и редактировать составные части текста.

Рассмотрим простой пример. Предположим, что имеется текстовый файл text1, содержащий строки

```
STRING 1
STRING 2
STRING 3
```

Предположим, нужно заменить символ `2` на слово `two`. Создадим командный файл demo_sed, содержащий строки

```
sed -e s/2/two/g text1 > tmp.sed
mv tmp.sed text1
```

Первая строка вызывает потоковый редактор `sed` с параметром `s` (от англ. *substitute*), далее следует выражение, подлежащее замене — `2`, затем указано выражение, которое нужно вставить — `two`. Параметр `g` указывает, что данная операция распространяется на содержимое всего файла. Для сохранения изменений нужно переназначить вывод `sed` в какой-либо временный файл (в данном случае tmp.sed), после чего использовать команду `mv`.

После выполнения командного файла text1 будет содержать такие строки:

```
STRING 1
STRING two
STRING 3
```

Рассмотрим более сложный пример. Предположим, что заменяемое слово и номер строки, в которой выполняется замена, вводятся с консоли, а сам файл имеет имя test и содержит строки:

```
STRING 1
STRING 2
STRING 3
STRING 4
STRING 5
```

Создадим командный файл demo_sed2, выполняющий такие операции.

Исходный текст командного файла представлен далее:

```
cat test
echo
echo Enter word you want substitute
echo Enter line which you want to substitute in
read word line
sed `expr $line`s/$word/xxx/ test  >  test.tmp
mv test.tmp test
```

Здесь слово, подлежащее замене, вместе с номером строки, в которой эта замена производится, вводятся с консоли посредством команды `read`. Следует обратить особое внимание на подстановки переменных в строке, где вызывается программа `sed`:

```
sed `expr $line`s/$word/xxx/ SED_test  >  SED_test.tmp
```

Обратите внимание, что выражение `` `expr $line` `` дает номер строки, `$word` — заменяемое слово, а заменяющим словом здесь является `xxx`.

Выполним командный файл:

```
# ./demo_sed2
STRING 1
STRING 2
STRING 3
STRING 4
STRING 5

Enter word you want substitute
Enter line which you want to substitute in
3 3
#
```

После выполнения командного файла строки файла test будут выглядеть так:

```
# cat test
STRING 1
STRING 2
STRING xxx
STRING 4
STRING 5
#
```

Подобным образом можно удалить строки из файла. Например, если файл test содержит строки:

```
STRING 1
STRING 2
STRING 3
STRING 4
```

то для удаления четвертой строки можно воспользоваться такими командными строками:

```
# sed 4d test > test.tmp
# mv test.tmp test
```

```
 mv: overwrite 'test'? y
#
```

Программа `sed` часто используется для редактирования значений переменных командного интерпретатора shell, например:

```
# str=r+s+t
# echo $str | sed s/+/-/g
r-s-t
```

В этом случае все символы `+` заменяются символами `-`.

* * *

На этом анализ возможностей командного интерпретатора можно закончить. Описаны далеко не все возможности командной оболочки, но даже из приведенных примеров видно, насколько полезным может оказаться этот инструментарий для пользователя системы UNIX.

Глава 5

Файловая система UNIX

Файловая система — наиболее важная и функционально самая сложная часть операционной системы UNIX, выполняющая организацию хранимых ресурсов операционной системы.

Особенностью UNIX является то, что все объекты операционной системы представляют собой файлы. Это означает, что файлами являются физические устройства, например, последовательный и параллельный порты, а также области дисковой и оперативной памяти, именованные каналы и сетевые соединения (сокеты). Такая унификация очень удобна, поскольку обеспечивает единый программный интерфейс для доступа к объектам, независимо от их реализации на физическом уровне. Например, системные вызовы `open()`, `read()`, `write()` и `close()` позволяют выполнять операции ввода/вывода единообразным способом как для файлов, находящихся на жестком диске, так и для физических портов ввода/вывода или программных каналов.

Естественно, в файловую систему входят собственно и файлы в том смысле, в каком мы привыкли их воспринимать — в виде наборов данных, содержащихся на жестких дисках или иных носителях.

Рассмотрим более подробно основные типы файлов, используемые в операционной системе. К ним следует отнести файлы, содержащие двоичные данные (бинарные файлы), файлы устройств, сокеты и именованные каналы, а также символические и жесткие ссылки.

Бинарные файлы — это наборы двоичных битов, содержащие тот или иной тип информации: текст, рисунки или программы, аудиоданные и т. д. Этот тип файлов наиболее широко используется, и когда говорят о файле, то обычно подразумевают именно такие объекты. Интерпретация данных, записанных в двоичные файлы, возлагается на операционную систему или другие программы. Например, когда говорят о текстовом файле, то подразумевают набор текстовых данных, содержащихся в этом файле и обрабатываемых

программой-редактором. Если говорят о программном файле, то это означает, что данные, записанные в такой файл, интерпретируются операционной системой как набор определенных команд, подлежащих выполнению. Хочу заметить, что бóльшая часть материала этой главы посвящена работе с файлами двоичных данных.

Файлы устройств позволяют операционной системе UNIX и другим программам взаимодействовать с аппаратными средствами и периферийными устройствами системы. Здесь хочу сделать важное замечание: нужно отличать файлы устройств от драйверов устройств. Управление конкретным физическим устройством осуществляется посредством специальной программы, которая называется драйвером устройства. Файлы устройств сами по себе не являются драйверами: их можно представить как шлюзы, через которые драйвер получает запросы.

Когда ядро получает запрос к файлу устройства, оно просто передает этот запрос соответствующему драйверу. Таким образом, файлы устройств позволяют программам взаимодействовать с драйверами ядра. По своей структуре они не являются файлами данных, но обрабатываются базовыми средствами файловой системы, а их характеристики записываются на диск. Взаимодействие пользовательской программы, файла устройства и драйвера показано на рис. 5.1. Здесь пользовательская программа обращается к устройству печати, подсоединенному к параллельному порту (ему может соответствовать файл устройства /dev/lp0).

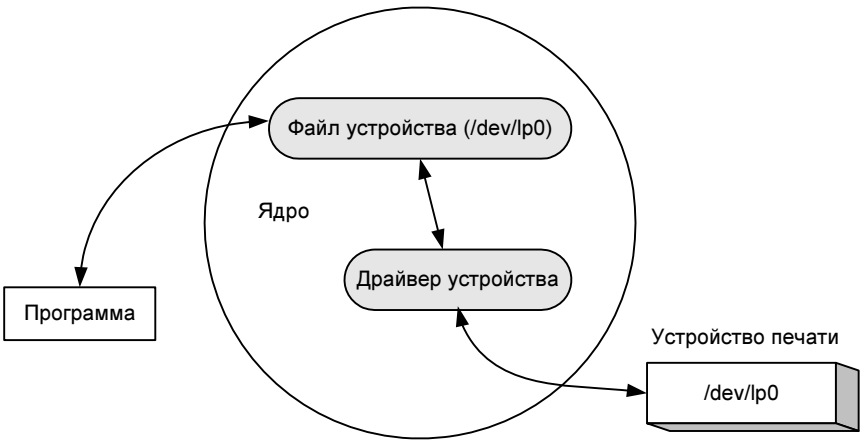


Рис. 5.1. Взаимодействие файла устройства, драйвера и программы

Файлы устройств, в свою очередь, можно разделить на два типа: файлы байт-ориентированных устройств и файлы блок-ориентированных устройств.

Файлы байт-ориентированных устройств позволяют связанным с ними драйверам выполнять собственную буферизацию ввода/вывода. Файлы блок-ориентированных устройств обрабатываются драйверами, которые осуществляют ввод/вывод большими блоками и возлагают обязанности по выполнению задач буферизации на ядро. Некоторые типы аппаратных средств, такие как накопители на жестких дисках и магнитных лентах, могут быть представлены файлами любого типа.

В системе может присутствовать несколько однотипных устройств, поэтому файлы устройств характеризуются двумя номерами: старшим и младшим. Старший номер устройства информирует ядро, к какому драйверу относится данный файл, а младший номер сообщает драйверу, к какому физическому устройству следует обращаться. Например, старший номер устройства 6 в Linux обозначает драйвер параллельного порта. Первый параллельный порт (/dev/lp0) будет иметь старший номер 6 и младший номер 0.

Некоторые драйверы используют младший номер устройства нестандартным способом. Например, драйверы накопителей на магнитных лентах часто руководствуются им при выборе плотности записи, а также определяют, нужно ли перемотать ленту после закрытия файла устройства. В некоторых системах драйвер терминала, управляющий последовательными устройствами, использует младшие номера устройств для идентификации модемов.

Рассмотрим еще один тип файла UNIX, который называют сокетом. Сокеты инкапсулируют соединения между процессами, позволяя им взаимодействовать, не подвергаясь влиянию других процессов. В UNIX поддерживается несколько типов сокетов, определяемых протоколом взаимодействия между процессами. Использование большинства сокетов предполагает наличие сети. Тем не менее, так называемые локальные или UNIX-сокеты можно использовать и для взаимодействия процессов на локальной машине. Должен заметить, что TCP-сокеты вполне подходят для межпроцессного взаимодействия, поскольку любая UNIX-система поддерживает интерфейс обратной связи (loopback interface) с сетевым адресом 127.0.0.1, который можно использовать при формировании соединения через сокеты.

Другие процессы, не участвующие во взаимодействии, распознают сокеты как элементы каталога, но выполнять операции чтения и записи над ними не могут. Взаимодействие процессов посредством сокетов очень популярно в самой операционной системе: с сокетами работают система печати, система X Window и система Syslog. Более детально взаимодействие процессов по сети мы будем рассматривать в последующих главах, сейчас же замечу, что сокеты создаются с помощью системного вызова `socket()`. При закрытии соединения с обеих сторон сокет можно удалить посредством команды `rm` либо системного вызова `unlink()`.

Еще один тип файлов, часто используемый, — именованные каналы. Подобно сокетам, именованные каналы обеспечивают межпроцессное взаимодействие на одной машине. Именованные каналы создаются командой `mknod`, а удаляются командой `rm`.

К отдельному типу объектов файловой системы можно отнести символические и жесткие ссылки, которые используются для обеспечения альтернативных способов обращения к файлам.

В современных файловых системах UNIX определен интерфейс уровня ядра, позволяющий работать с различными аппаратными интерфейсами. При этом часть файлов обрабатывается традиционной дисковой подсистемой, другие управляются отдельными драйверами ядра, как в случае сетевых файловых систем, где используется драйвер, перенаправляющий запросы серверу на другой компьютер.

Большинство операционных систем UNIX поддерживает несколько типов файловых систем. Кроме базовой версии, берущей начало от 4.3BSD, поддерживаются и другие файловые системы, например, обладающие повышенной надежностью или упрощенными средствами восстановления после сбоев (VXFS в HP-UX), а также системы, поддерживающие другую семантику (Solaris), и системы, построенные на других типах носителей (жесткие диски DOS или компакт-диски стандарта ISO-9660).

Большинство современных файловых систем имеет определенные отличия от базовой версии, поэтому в этой главе описывается обобщенный тип файловой системы.

Файловую систему можно рассматривать, с одной стороны, как логическую структуру, в виде дерева каталогов и файлов, с четко установленной иерархией. Именно в таком виде и представляется файловая система UNIX пользователю. С другой стороны, файловая система — это совокупность расположенных на физическом носителе упорядоченных и неупорядоченных двоичных данных. Пользователь обычно не имеет доступа непосредственно к блокам данных на носителях, хотя и может управлять ими при помощи программ, в которых используются соответствующие системные вызовы. Управление физической файловой системой — прерогатива функций ядра, поэтому вмешательство в этот процесс очень опасно, даже если вы хорошо представляете себе, что делаете.

В этой книге мы будем иметь дело в основном с объектами логической файловой системы, тем не менее, для лучшего понимания работы вначале кратко ознакомимся с организацией физической файловой системы.

Файловые системы обычно располагаются на жестких дисках, каждый из которых состоит из одной или нескольких логически связанных групп цилинд-

ров, называемых разделами (partitions). Физическое расположение и размер раздела устанавливаются при форматировании диска. В операционных системах UNIX разделы являются независимыми устройствами, доступ к которым осуществляется как к различным носителям данных. Один раздел содержит, как правило, только одну физическую файловую систему.

Для операционных систем UNIX разработано много типов файловых систем (ufs, ext2, vxfs и т. д.), каждая из которых имеет свои особенности. Операционная система и функционирующие программы пользователей работают не с физическими блоками данных, а с логическими структурами данных, которые образуют логическую файловую систему.

Логическая файловая система включает в себя следующие компоненты:

- суперблок (superblock) — область на диске, содержащая общую информацию о файловой системе;
- массив индексных дескрипторов (ilist) — здесь хранится общая информация (метаданные) обо всех файлах файловой системы. Размер массива индексных дескрипторов является фиксированным и задается при создании физической файловой системы;
- индексный дескриптор (inode) — содержит информацию о статусе файла и указывает на расположение данных этого файла. Ядро обращается к индексному дескриптору по индексу в массиве ilist. Один из дескрипторов является корневым для физической файловой системы, обеспечивая доступ к структуре каталогов и файлов после монтирования файловой системы;
- блоки хранения данных — содержат данные обычных файлов и каталогов. Доступ к данным файла осуществляется через индексный дескриптор, хранящий ссылки на блоки данных.

Рассмотрим компоненты логической файловой системы более подробно и начнем с суперблока. Он содержит информацию, используемую при монтировании и управлении файловой системой. Любая файловая система содержит только один суперблок, который расположен в начале раздела. При монтировании файловой системы суперблок считывается в память ядра системы и находится там до ее отключения (демонтирования).

В суперблоке содержится информация о:

- типе файловой системы;
- размере файловой системы в логических блоках, включая сам суперблок, массив индексных дескрипторов и блоки хранения данных;
- размере массива индексных дескрипторов;
- количестве свободных блоков;

- количестве свободных индексных дескрипторов;
- флагах;
- размере логического блока файловой системы (512, 1024, 2048, 4096, 8192);
- номерах (в виде списков) свободных индексных дескрипторов;
- адресах (в виде списков) свободных блоков.

Количество свободных индексных дескрипторов и блоков хранения данных может быть большим, поэтому хранение двух последних списков в суперблоке нерационально. В списке свободных индексных дескрипторов хранится только часть информации. Если число свободных дескрипторов приближается к нулю, ядро просматривает список и формирует новый список свободных дескрипторов.

Такой подход неприемлем для свободных блоков хранения данных, поскольку по содержимому блока нельзя определить, занят он или свободен, поэтому список адресов свободных блоков хранится целиком. Список адресов свободных блоков может занимать несколько блоков хранения данных, но суперблок содержит только один блок этого списка. Первый элемент этого блока указывает на блок, хранящий продолжение списка.

Выделение свободных блоков для размещения файла осуществляется с конца списка суперблока. Если в списке остается единственный элемент, ядро интерпретирует его как указатель на блок, содержащий продолжение списка. В этом случае содержимое данного блока считывается в суперблок, и блок становится свободным.

При таком подходе можно использовать дисковое пространство под списки, размеры которых пропорциональны свободному месту в файловой системе. Если свободное место практически отсутствует, то список адресов свободных блоков целиком помещается в суперблоке.

Важным компонентом логической файловой системы являются индексные дескрипторы. Индексный дескриптор содержит информацию (метаданные) о файле, необходимую для обработки данных. Любой файл ассоциируется с одним индексным дескриптором, хотя может иметь несколько имен (жестких ссылок) в файловой системе, каждое из которых указывает на один и тот же индексный дескриптор.

В индексном дескрипторе содержится:

- номер;
- тип файла;
- права доступа к файлу;
- количество ссылок на файл в каталогах;

- ❒ идентификатор пользователя и группы-владельца;
- ❒ размер файла в байтах;
- ❒ время последнего доступа к файлу;
- ❒ время последнего изменения файла;
- ❒ время последнего изменения индексного дескриптора файла;
- ❒ указатели на блоки данных файла (обычно 10);
- ❒ указатели на косвенные блоки (обычно 3).

В то же время в индексном дескрипторе отсутствует информация:

- ❒ об имени файла (оно находится в блоках хранения данных каталога);
- ❒ о содержимом файла (размещается в блоках хранения данных).

Размер индексного дескриптора обычно составляет 128 байтов. Индексные дескрипторы поддерживают множественные уровни косвенности, что позволяет отслеживать файлы больших размеров, не растрачивая при этом дисковое пространство для небольших файлов. Если структура файловой системы изменяется, то выполняется синхронизация файловой системы.

Если файл открывается для выполнения какой-либо операции, ядро помещает копию дискового индексного дескриптора в соответствующую таблицу индексных дескрипторов, расположенную в памяти. После этого все последующие изменения индексного дескриптора происходят в памяти. Измененная структура файловой системы записывается на диск только при выполнении специальной команды `sync`.

Если происходит внезапное прекращение работы системы, то структура суперблока и массива индексных дескрипторов на диске может не соответствовать структуре блоков данных (inconsistency). В таких случаях потребуется восстановление файловой системы специальными программами, например, утилитой `fsck`.

Несколько слов о терминологии. Очень часто при использовании термина "файловая система" подразумевается, что речь идет о логической файловой системе. В дальнейшем, если не будет особо оговорено, эти термины мы будем употреблять как синонимы.

Конечный пользователь вряд ли смог бы эффективно работать с объектами файловой системы, если бы имел дело с логическими структурами, поэтому для конечного пользователя операционная система UNIX использует интерфейс высокого уровня, в котором файловая система представляется как единая иерархическая структура, организованная в виде дерева каталогов.

Основой любой файловой системы является корневой каталог (обозначается как /). Все остальные каталоги и файлы располагаются в рамках структуры, порожденной корневым каталогом (в нем и в его подкаталогах), независимо от их физического местонахождения. Замечу, что каталог является обычным файлом, содержащим информацию о других файлах. Он позволяет четко структурировать объекты файловой системы. Структуру файловой системы UNIX можно представить себе так, как показано на рис. 5.2.

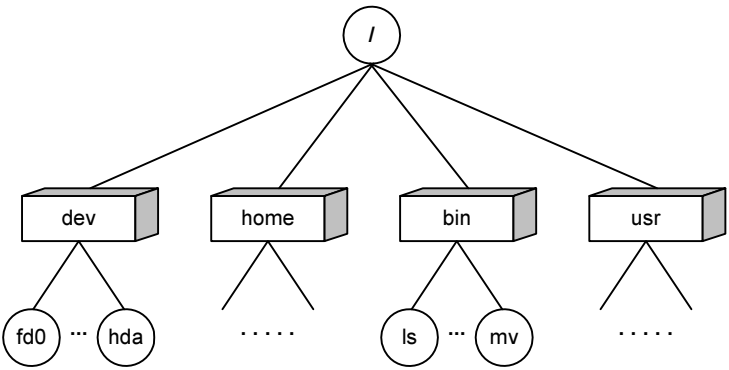


Рис. 5.2. Структура файловой системы UNIX

Цепочка имен каталогов, через которые необходимо пройти для доступа к заданному файлу, вместе с именем этого файла называется путевым именем. Путевые имена могут быть абсолютными (например, /tmp/afile) или относительными (например, book3/filesystem). Последние интерпретируются, начиная с текущего каталога. На имена файлов и каталогов накладываются определенные ограничения: имя каталога должно состоять не более чем из 255 символов, а в отдельном путевом имени не должно быть более 1023 символов.

Для корневого каталога обязательно создается отдельная физическая файловая система, а сам он является точкой ее монтирования, о чем свидетельствует наличие подкаталога lost+found. Корневой каталог должен быть всегда доступен и монтируется автоматически при запуске системы. Следует сказать, что при такой структуре все остальные физические файловые системы с формальной точки зрения для функционирования операционной системы UNIX не нужны.

В большинстве операционных систем используется более-менее стандартная структура каталогов файловой системы, при этом все каталоги имеют предо-

пределенные назначения (хотя соблюдать это вовсе не обязательно). Назначение каталогов приводится далее:

- /bin — содержит пользовательские выполняемые программы — сейчас обычно является символической ссылкой, указывающей на /usr/bin;
- /dev — каталог для специальных файлов устройств — может иметь подкаталоги для различных классов и типов устройств, например, dsk, rdsk, rmt, inet (в System V);
- /etc — каталог для конфигурационных файлов — может иметь подкаталоги для различных компонентов и служб (конфигурационные файлы в UNIX — обычные текстовые файлы);
- /home — каталог для размещения начальных каталогов пользователей — часто является точкой монтирования отдельной физической файловой системы;
- /lib — каталог для библиотек — сейчас обычно является символической ссылкой, указывающей на /usr/lib;
- /lost+found — подкаталог, имеющийся в каждом каталоге, являющемся точкой монтирования физической файловой системы на диске;
- /mnt — точка монтирования для файловых систем на съемных носителях или дополнительных дисках (может содержать подкаталоги для отдельных типов носителей, например, CD-ROM или floppy, может быть пустой);
- /opt — каталог для дополнительного коммерческого программного обеспечения (может быть пустым или отсутствовать (в BSD-системах));
- /proc — каталог псевдофайловой системы, предоставляющей в виде каталогов и файлов информацию о ядре, памяти и процессах, работающих в системе;
- /sbin — каталог для системных выполняемых программ, необходимых для решения задач системного администрирования;
- /tmp — каталог для временных файлов — имеет установленный sticky-бит (от англ. *sticky* — липучка) и доступен для записи и чтения всем пользователям, обычно создается в виде отдельной физической файловой системы, в том числе в виртуальной памяти;
- /usr — в этом каталоге находятся выполняемые программы, библиотеки, заголовочные файлы, справочные руководства (/usr/share/man), исходные тексты ядра и утилит системы (в Linux), файлы очереди печати (/usr/spool в BSD-системах) и т. д. Часто каталог является точкой монтирования

отдельной физической файловой системы. Далее представлены основные его подкаталоги:

- /usr/bin — основные выполняемые программы и утилиты;
- /usr/include — заголовочные файлы библиотек; может содержать подкаталоги;
- /usr/lib — статически и динамически компонуемые библиотеки; может содержать подкаталоги;
- /usr/local — каталог для дополнительного свободно распространяемого программного обеспечения, содержит структуру подкаталогов, аналогичную корневому каталогу (bin, etc, include, lib и т. д.);

❒ /var — в System V и Linux этот каталог является аналогом каталога, используемого для хранения файлов различных сервисных подсистем (/usr/spool), например, файлов журналов системы. Так, основной журнал системы, управляемый демоном syslogd, размещается в виде нескольких файлов в подкаталоге /var/adm. Там же, в файле /var/adm/messages, сохраняются сообщения времени загрузки. Имеет смысл создавать отдельную физическую файловую систему для размещения этого каталога.

Наличие, назначение и использование других каталогов верхнего уровня и подкаталогов зависит от версии операционной системы UNIX, установленного системного и прикладного программного обеспечения и конфигурации системы, созданной пользователем или системным администратором.

В операционной системе UNIX путевое имя файла принято называть жесткой ссылкой. Большинство файлов в UNIX имеет всего одну жесткую ссылку. Однако при помощи команды `ln` пользователь может создавать дополнительные жесткие ссылки на файлы.

Пусть имеется путевое имя файла /home/user1/user1.text. Тогда, после выполнения команды

```
ln /home/user1/user1.text /home/user1/text.new
```

к этому же файлу можно обращаться как к /home/user1/text.new.

Ссылка, которую называют символической или "мягкой", обеспечивает возможность вместо путевого имени файла указывать псевдоним. Когда ядро обнаруживает символическую ссылку при поиске файла, оно извлекает из нее хранящееся в ней путевое имя. Различие между жесткими и символическими ссылками состоит в том, что жесткая ссылка — прямая, т. е. указывает непосредственно на индексный дескриптор файла, тогда как символическая ссылка указывает на файл по имени. Файл, адресуемый символической ссылкой, и сама ссылка физически являются разными объектами файловой системы.

Символические ссылки создаются командой `ln -s`, а удаляются командой `rm`. Поскольку они содержат произвольные путевые имена, то могут указывать на файлы, хранящиеся в других файловых системах, и даже на несуществующие файлы. Иногда несколько символических ссылок образуют цикл.

Для файла /home/user1/user1.text вместо жесткой ссылки можно создать символическую:

```
ln -s /home/user1/user1.text /home/user1/text.new
```

Обозначение `..` в путевых именах, включающих символические ссылки, лучше не использовать, поскольку по символическим ссылкам нельзя проследовать в обратном направлении.

До сих пор мы рассматривали файловую систему, не выдвигая никаких предположений о том, каким образом операционная система делает доступными ресурсы файловой системы для пользователя. Все современные операционные системы скрывают детали операций с файловой системой от пользователя, предоставляя ему полностью функциональную логическую файловую систему в виде дерева каталогов. Все основные операции по инициализации физических файловых систем, распределению дискового пространства, форматированию дисков и их подключению к системе выполняются, как правило, в процессе инсталляции операционной системы UNIX и в большинстве случаев осуществляются автоматически или при минимальном вмешательстве пользователя.

Тем не менее, нередко возникают ситуации, когда пользователь должен выполнить определенные манипуляции с файловой системой вручную, например, если требуется подключить или отключить файловую систему или же изменить параметры подключения для нее. Другой случай, весьма неприятный — нарушение целостности файловой системы, в результате чего возникают ошибки при выполнении файловых операций, что требует восстановления файловой системы. Во всех подобных случаях понимание процессов, происходящих при манипулировании файловой системой, просто необходимо для правильных действий. Сейчас мы рассмотрим более подробно подключение и отключение файловых систем, а также контроль их функционирования.

5.1. Подключение, отключение и восстановление файловых систем

Перед использованием файлов система UNIX должна выполнить процедуру подключения или, в терминах UNIX, монтирования файловой системы. Жесткий диск может содержать один или несколько логических разделов, на которые он разбит дисковым драйвером, причем каждому разделу соответст-

вует файл устройства с определенным именем. Процессы обращаются к данным раздела, открывая соответствующий файл устройства и затем выполняя запись и чтение из него.

Раздел диска может содержать логическую файловую систему, состоящую из блока начальной загрузки, суперблока, списка индексов и информационных блоков. Системная команда `mount` (монтировать) связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем, а команда `umount` (демонтировать) выключает файловую систему из иерархии. Таким образом, команда `mount` дает пользователям возможность обращаться к данным в дисковом разделе как к файловой системе, а не как к последовательности дисковых блоков. Все физические файловые системы, кроме корневой (`/`), считаются съемными в том смысле, что они могут быть как доступны для пользователей, так и не доступны.

Команда `mount` сообщает системе, что блочное устройство или удаленный ресурс доступен для пользователей в точке монтирования, которая к моменту монтирования уже должна существовать. В этом случае точка монтирования становится корневым каталогом вновь смонтированного устройства или ресурса.

Таким образом, команда `mount` монтирует физическую файловую систему или ресурс к общей логической файловой системе.

Данная команда может иметь следующий синтаксис:

```
mount [-v | -p]
mount [-F ФС] [-V] [-o опции]{устройство|точка_монтирования}
mount [-F ФС] [-V] [-o опции]{устройство точка_монтирования}
```

Команда `mount` при указанных во время вызова аргументах проверяет их, за исключением устройства *`устройство`*, и вызывает соответствующий модуль монтирования для указанного типа файловой системы. Вызванная без аргументов команда `mount` отображает список всех смонтированных файловых систем, считывая информацию из соответствующей таблицы. Если указан неполный список аргументов (например, указаны только устройства или точки монтирования, или указаны оба эти аргумента, но не задан тип файловой системы), команда `mount` просматривает таблицу стандартных файловых систем в поисках недостающих аргументов, после чего вызывает соответствующий модуль монтирования для соответствующего типа файловой системы.

Обратная процедура по отношению к монтированию называется демонтированием и выполняется командой `umount` со следующим синтаксисом:

```
umount [-V] [-o опции]{устройство|точка_монтирования}
```

В большинстве систем занятую файловую систему демонтировать невозможно. В ней не должно быть открытых файлов и выполняющихся процессов. Если демонтируемая файловая система содержит исполняемые программы, они не должны быть запущены. Для большинства типов файловых систем нет специфического модуля демонтирования. Если такой модуль существует, он выполняется, иначе файловая система демонтируется стандартным модулем.

Команды `mount` и `umount` могут работать со следующими основными опциями:

- `-v` — дополнительно отображаются тип файловой системы и флаги. Поля *`точка_монтирования`* и *`устройство`* переставлены;
- `-p` — отображает список смонтированных файловых систем в формате таблицы смонтированных файловых систем;
- `-F` — задает тип файловой системы для монтирования. Тип файловой системы должен быть либо задан, либо определяется по таблице стандартных файловых систем в ходе монтирования;
- `-V` — отображает результирующую командную строку, но не выполняет команду. Командная строка генерируется с помощью опций и аргументов, указанных пользователем, путем добавления к ним, при необходимости, информации, взятой из таблицы стандартных файловых систем;
- `-o` — задает специфические опции для указанного типа физической файловой системы.

Любой пользователь может вызывать команду `mount` для получения списка смонтированных файловых систем и ресурсов. Например:

```
# mount
/dev/hda8 on /              type ext3     (rw)
none      on /proc          type proc     (rw)
usbdevfs  on /proc/bus/usb  type usbdevfs (rw)
/dev/hda7 on /boot          type ext3     (rw)
none      on /dev/pts       type devpts   (rw,gid=5,mode=620)
none      on /dev/shm       type tmpfs    (rw)
```

Монтирование и демонтирование файловых систем разрешено только суперпользователю root.

Команда `mount` добавляет запись в таблицу смонтированных файловых систем, а `umount` удаляет запись из этой таблицы. Поля в таблице смонтированных устройств разделены пробелами и предоставляют информацию о:

- типе блочного специального устройства;
- точке монтирования;
- типе смонтированной файловой системы;

- опциях монтирования;
- времени, когда файловая система была смонтирована.

Для корректного выполнения операций монтирования/демонтирования необходима дополнительная информация, которая находится в таблице стандартных файловых систем (файл /etc/vfstab или /etc/fstab, в зависимости от реализации UNIX). В соответствующих полях этой таблицы, разделенных пробелами или символами табуляции, описаны стандартные параметры для физических файловых систем:

- специальное блочное устройство или имя монтируемого ресурса;
- неформатированное (специальное символьное) устройство для проверки утилитой `fsck`;
- стандартный каталог монтирования;
- тип файловой системы;
- числовой параметр, используемый командой `fsck` для принятия решения об автоматической проверке файловой системы и о порядке этой проверки по отношению к другим файловым системам;
- признак автоматического монтирования файловой системы;
- опции монтирования.

Вот как может выглядеть такая таблица в операционной системе Linux (она называется fstab):

```
# cat fstab
LABEL=/      /           ext3        defaults              1 1
LABEL=/boot  /boot       ext3        defaults              1 2
none         /dev/pts    devpts      gid=5,mode=620        0 0
none         /proc       proc        defaults              0 0
none         /dev/shm    tmpfs       defaults              0 0
/dev/hda9    swap        swap        defaults              0 0
/dev/cdrom   /mnt/cdrom  udf,iso9660 noauto,owner,kudzu,ro 0 0
/dev/fd0     /mnt/floppy auto        noauto,owner,kudzu    0 0
```

5.2. Контроль дискового пространства

В процессе функционирования операционной системы UNIX любой пользователь рано или поздно сталкивается с проблемой нехватки дискового пространства. Даже если в системе установлены дисковые накопители большой емкости, наступает момент, когда места на дисках не хватает. Современное программное обеспечение использует, как правило, значительные ресурсы

постоянной памяти, сохраняя при этом устойчивую тенденцию к дальнейшему увеличению такого потребления.

Для контроля используемых ресурсов файловой системы в UNIX предусмотрены специальные утилиты, такие как `du` и `df`.

Команда `df` позволяет получить информацию о смонтированных файловых системах. Она выдает отчет о доступном и использованном дисковых пространствах на файловых системах. Синтаксис команды может быть представлен как

`df [`*`опции`*`] [`*`файл`*`]`

Здесь опции и параметры определяют формат отображаемой информации о файловых системах. При запуске без аргументов `df` выдает отчет по доступному и использованному пространству для всех смонтированных файловых систем (всех типов). Запущенная с опцией `-k` команда `df` отображает размеры файловых систем в килобайтах. В этом случае информация о каждой физической файловой системе отображается в отдельной строке и включает в себя:

- ❒ имя специального файла или смонтированного ресурса;
- ❒ общий и используемый объем дискового пространства;
- ❒ объем, доступный для использования обычными пользователями;
- ❒ процент свободного дискового пространства в файловой системе;
- ❒ точку монтирования.

Если в качестве параметра команды `df` задано имя файла, то отображается отчет по файловой системе, которая его содержит. При этом если параметр *`файл`* является дисковым файлом устройства, содержащим смонтированную файловую систему, то отображается доступное пространство для этой файловой системы, а не той, где содержится файл устройства. Команда `df` поддерживается стандартами POSIX и GNU, при этом некоторые параметры являются специфичными для каждого из стандартов.

Для команды `df` в стандарте POSIX все размеры выдаются в блоках по 512 байтов, но если задана опция `-k`, то используются блоки размером по 1024 байта. Формат вывода не стандартизован, кроме случая, когда используется опция `-P`. Кроме того, если файл является каталогом или именованным каналом (FIFO), то результат не определен.

Для команды `df` в стандарте GNU все размеры отображаются в блоках по 1024 байта (если размер блока не задан), за исключением случая, когда установлена переменная `POSIXLY_CORRECT`: тогда размер блока соответствует POSIX-версии этой команды. В стандарте GNU используются опции:

`[-ahHiklmPv]`

`[-t` *`тип_файловой_системы`*`]`

```
[-x тип_файловой_системы]
[--block-size=размер]
[--print-type]
[--no-sync]
[--sync]
[--help]
[--version]
```

Рассмотрим более подробно некоторые опции:

- `-k` — используется размер блока в 1024 байта вместо размера по умолчанию 512 байтов;
- `-P` — вывод осуществляется в шесть колонок, с заголовком "Filesystem N-blocks Used Available Capacity Mounted on". Размер блока устанавливается по умолчанию (512 байтов) или 1024 байта при задании опции `-k`;
- `-a`, `--all` — включает в список вывода файловые системы, имеющие размер 0 блоков (не выводятся по умолчанию);
- `--block-size=`*размер* — отображает размер в блоках, размер в байтах которых задан;
- `-k`, `--kilobytes` — отображает выводимую информацию в блоках по 1024 байта;
- `-l`, `--local` — отображает данные только о локальных файловых системах;
- `-t` *тип_файловой_системы*, `--type=`*тип_файловой_системы* — показывает только файловые системы с указанным типом, при этом разрешается задавать несколько типов файловых систем с использованием нескольких опций `-t`. По умолчанию отображается информация обо всех файловых системах.

Рассмотрим несколько примеров использования команды `df`:

```
$ df -k
Filesystem     1K-blocks      Used      Available  Use%  Mounted on
/dev/hda5      7091968        2004488   4727224    30%   /
/dev/hda3      101107         14832     81054      16%   /boot
none           256900         0         256900     0%    /dev/shm
```

В этой команде используется опция `-k`, поэтому вывод на консоль выполняется в пересчете на однокилобайтовые блоки.

В следующем примере задается опция `-a`, поэтому на дисплей выводится список всех файловых систем, включающий специальные файловые системы:

```
$ df -a
Filesystem    1K-blocks   Used      Available  Use%  Mounted on
```

```
/dev/hda5      7091968     2004488     4727224     30%   /
none           0           0           0           -     /proc
usbdevfs       0           0           0           -     /proc/bus/usb
/dev/hda3      101107      14832       81054       16%   /boot
none           0           0           0           -     /dev/pts
none           256900      0           256900      0%    /dev/shm
```

Заданная без опций, команда `df` может отображать информацию в таком виде, как показано в примере:

```
# df
/                  (/dev/dsk/c0d0s0 ):    450540 blocks   120616 files
/usr               (/dev/dsk/c0d0s3 ):   2627968 blocks   338652 files
/boot              (/dev/dsk/c0d0p0:boot): 18350 blocks       -1 files
/proc              (/proc           ):         0 blocks     3615 files
/dev/fd            (fd              ):         0 blocks        0 files
/etc/mnttab        (mnttab          ):         0 blocks        0 files
/var               (/dev/dsk/c10d0s1):    574236 blocks   240784 files
/var/run           (swap            ):    647568 blocks    43108 files
/tmp               (swap            ):    647568 blocks    43108 files
/home              (/dev/dsk/c10d0s4):   2569298 blocks   379999 files
/fs                (/dev/dsk/c20d0s1):   1116738 blocks   688872 files
```

Команда `du` позволяет вывести на консоль отчет об использовании дискового пространства заданными файлами, а также каждым каталогом в иерархии подкаталогов для каждого указанного каталога. Имейте в виду, что команда `du` показывает не фактический размер файла в байтах, а отображает только количество выделенных блоков или байтов на диске.

Размер блока зависит от настройки соответствующих параметров операционной системы и изменяется в широком диапазоне. Обычно используют блоки размером в 1 Кбайт, а минимальное дисковое пространство, выделяемое для файла и каталога, устанавливают равным 4 блока или 4 Кбайт (для блока размером в 1 Кбайт). Такая группа блоков называется кластером. Если для файла выделено 12 Кбайт дискового пространства, то говорят, что он занимает 3 кластера. Размер кластера, в свою очередь, также может варьироваться (обычно он равен 4 блокам). Очень часто возникает ситуация, когда для файлов, имеющих разные размеры, выделяется одинаковое дисковое пространство. Например, для файлов с размерами 2 байта и 2 Кбайт будет выделено на жестком диске одинаковое пространство в 4 Кбайт, т. е. 1 кластер.

Рассуждая о свободном дисковом пространстве, нужно учитывать тот факт, что операционная система располагает свободным дисковым пространством,

пересчитанным на количество свободных кластеров. Это означает, что фактический размер файлов на диске и выделенное под них дисковое пространство никогда не совпадают в принципе, поэтому свободное дисковое пространство определяется не количеством свободных байтов, а количеством свободных кластеров.

Эту особенность следует учитывать при работе с файловой системой UNIX. Именно поэтому размеры блока и кластера влияют на многие параметры функционирования операционной системы, особенно на ее быстродействие.

Следует учитывать, что операционная система работает с файлами быстрее, если используются кластеры больших размеров, однако в этом случае уменьшается свободное дисковое пространство, что необходимо учитывать при настройке UNIX.

Если команда `du` выполняется без аргументов, то на консоли отображается отчет о дисковом пространстве для текущего каталога. Команда поддерживает опции как стандарта POSIX, так и GNU. Вот некоторые опции стандарта POSIX и их смысл:

- `-a` — отображает размеры для всех файлов;
- `-k` — использует размер блока 1024 байта вместо 512 по умолчанию;
- `-s` — отображает информацию только для указанных явно аргументов, а не для их подкаталогов;
- `-x` — вычисляет размеры только для той файловой системы, где расположен объект, заданный параметром.

Команда `du` может принимать и опции стандарта GNU:

- `-a`, `--all` — показывает размеры для всех файлов;
- `-b`, `--bytes` — отображает размеры в байтах вместо килобайтов;
- `--block-size=`*`количество`* — отображает размеры в блоках размером *`количество`* байтов;
- `-c`, `--total` — отображается итоговая информация по всем параметрам после того, как они будут обработаны. Эта возможность может быть использована для подсчета суммарного использованного дискового пространства для всего списка заданных файлов и каталогов;
- `--exclude=`*`шаблон`* — при рекурсивном выполнении пропускает файлы или каталоги, чьи имена совпадают с заданным шаблоном. Этот шаблон может быть любым файловым шаблоном bash;
- `-k`, `--kilobytes` — отображает размеры в килобайтах (1024 байта);

- `-m`, `--megabytes` — отображает размеры в мегабайтах (1 048 576 байтов);
- `-s`, `--summarize` — отображает только суммарный итог для каждого аргумента;
- `-S`, `--separate-dirs` — выдает размер каждого каталога в отдельности, не включая размеры подкаталогов;
- `-x`, `--one-file-system` — пропускает каталоги, находящиеся в другой файловой системе.

Размеры указываются в блоках по 1024 байта (если размер не задан посредством опций), за исключением случая, когда задана переменная окружения `POSIXLY_CORRECT`. Тогда размер блока соответствует версии POSIX.

Команда `du` без параметров показывает размер дискового пространства, занимаемого файлом или каталогом в целом:

```
$ du $HOME/DIR1
12      /home/yury/DIR1/USR1
24      /home/yury/DIR11
```

Результат выдается в блоках, размер которых определяется UNIX. В данном примере информация о занимаемом дисковом пространстве рассчитывается, исходя из размера блока в 1 Кбайт. Для опредения объема дискового пространства при другом размере блока нужно использовать опцию `-B`:

```
$ du -B 512 $HOME/DIR1
24      /home/yury/DIR/USR1
48      /home/yury/DIR1
```

Здесь используется блок размером в 512 байтов. Для получения размера занимаемого дискового пространства в байтах можно выполнить команду `du` с опцией `-b`:

```
$ du -b $HOME/DIR1
12288   /home/yury/DIR1/USR1
24576   /home/yury/DIR1
```

Тот же самый результат получается, если выполнить команду `du` с опцией `-B` и параметром `1`. Например, для каталога $HOME/DIR1/USR1 команды, показанные далее, дают один и тот же результат:

```
$ du -B 1 $HOME/DIR1/USR1
12288   /home/yury/DIR1/USR1

$ du -b $HOME/DIR1/USR1
12288   /home/yury/DIR1/USR1
```

Если команда `du` задана с опцией `-c`, то кроме информации по отдельным каталогам выдается суммарный размер используемого дискового пространства:

```
$ du -c $HOME/DIR1
12      /home/yury/DIR1/USR1
24      /home/yury/DIR1
24      total
```

Для сравнения можно выполнить команду `ls`, которая показывает фактический размер файлов в байтах:

```
$ ls -l $HOME/DIR1
total 12
-rw-rw-r--    1 yury     yury           19 Dec 14 10:35 tmp
drwxrwxr-x    2 yury     yury         4096 Dec 14 16:26 USR1
-rw-rw-r--    1 yury     yury          120 Dec 14 16:27 test1
$ ls -l $HOME/DIR1/USR1
total 8
-rw-rw-r--    1 yury     yury           28 Dec 14 16:26 test2
-rw-rw-r--    1 yury     yury           57 Dec 14 16:26 test3
```

Легко проследить разницу между суммарным размером файлов (в байтах), находящихся в каталоге $HOME/DIR1, и размером дискового простанства, занимаемого этими файлами. Суммарный размер файлов равен 224 байта, в то время как для этих же файлов выделено 20 Кбайт дискового пространства.

Для того чтобы просмотреть размер файлов, необходимо ввести полное путевое имя файла. Следующие примеры демонстрируют применение команды `du` для работы с файлами:

```
$ du $HOME/DIR1/tmp
4       /home/yury/DIR1/tmp
$ du -b $HOME/DIR1/tmp
4096    /home/yury/DIR1/tmp
$ du -B 512 $HOME/DIR1/tmp
8       /home/yury/DIR1/tmp
```

Команда `du`, примененная с опцией `--exclude`, позволяет исключить из рассмотрения отдельные файлы:

```
$ du -b --exclude 't*' $HOME/DIR1
4096    /home/yury/DIR1/USR1
16384   /home/yury/DIR1
```

В этом примере из результата выполнения команды исключены данные по файлам, удовлетворяющим шаблону `t*`.

Команду `du` можно применить в командных файлах. Следующий простой пример позволяет получить информацию об использовании дискового пространства в нескольких каталогах:

```
echo Size in bytes of directories
for x in $*
 do
  du -b $x
 done
```

Здесь выполняется обработка нескольких каталогов, имена которых указываются в командной строке.

Перед подключением файловой системы необходимо, чтобы она существовала на физическом диске, либо самому ее создавать. Почти все реализации операционной системы UNIX позволяют создавать файловые системы автоматически, без участия пользователя в процессе инсталляции. Тем не менее, могут возникнуть ситуации, требующие создания файловых систем вручную, например, при невозможности восстановления файловой системы после повреждения. Кроме того, знание, как создается файловая система, может помочь при анализе ее функционирования. Сейчас мы рассмотрим некоторые наиболее важные аспекты создания файловых систем.

В большинстве UNIX-систем для создания файловых систем используется команда `mkfs`. Для создания файловой системы в командной строке указывается ее тип (`ФС`), а также целый ряд специальных опций и операндов. Вот синтаксис команды:

```
mkfs [-F ФС][-V][-m] [-o опции] устройство размер [операнды]
```

Задаваемые опции и операнды зависят от типа создаваемой файловой системы. Основные из них представлены далее:

- `-F` — определяет тип создаваемой файловой системы. Он задается либо в командной строке, либо берется из файла, содержащего таблицу стандартных файловых систем (/etc/vfstab в System V, /etc/fstab в других версиях UNIX);
- `-V` — отображает введенную командную строку без выполнения самой команды. Напомню, что командная строка создается как при помощи опций, указанных пользователем, так и путем добавления информации, взятой из таблицы стандартных файловых систем. Эта опция позволяет проверить корректность командной строки;
- `-m` — отображает командную строку, использованную для создания файловой системы, при этом файловая система уже должна существовать. Эта опция неприменима с другими параметрами;

- `-o` — задает опции, специфические для указанного типа физической файловой системы, например, специальное символьное устройство, на котором будет создана файловая система, или размер файловой системы в 512-байтовых блоках.

До сих пор мы рассматривали файловые системы в предположении, что они функционируют нормально. К сожалению, иногда файловые системы повреждаются. Это вызывается разными причинами, в числе которых можно назвать износ физического носителя информации, неисправность электронных схем или, нередко, случайное разрушение логических структур файловой системы (суперблок, таблица индексных дескрипторов и т. д.) самим пользователем.

Проблема целостности (consistency) файловой системы является основой корректной работы операционной системы и ее надежности. В самой операционной системе UNIX предусмотрено несколько операций, обеспечивающих более надежную работу файловой системы. Так, например, во время записи файла на жесткий диск его индексный дескриптор и блоки устанавливаются в определенном порядке, при этом такую операцию называют "упорядочением записи". Одним из способов повышения надежности является периодическая запись системных буферов на жесткий диск, известная как "автоматическая модификация". Кроме этого, все современные системы UNIX в процессе работы выполняют определенные процедуры диагностирования.

Тем не менее, целиком полагаться на средства проверки операционной системы не стоит. Будет лучше, если пользователь протестирует файловую систему при возникновении сомнений в ее корректной работе. Повысить надежность работы файловой системы можно и в том случае, если соблюдать некоторые простые правила. Вот они:

- при выключении компьютера используйте стандартные для этого случая команды операционной системы, например, `shutdown`. Это обеспечивает корректное закрытие всех файлов и демонтирование всех файловых систем;
- наиболее важные данные записывайте на альтернативный носитель, например, CD;
- время от времени проверяйте целостность файловой системы с помощью встроенных программных средств, например, утилиты `fsck`.

Программа `fsck` включена во все версии операционных систем и очень полезна при проверке целостности файловых систем. При подозрении, что что-то не в порядке с файловой системой, всегда используйте `fsck`. Утилита позволяет найти и исправить повреждения (inconsistency) в файловых системах, причем допускает два режима работы — автоматический или интерактивный.

При значительных повреждениях файловой системы программу `fsck` лучше запускать в интерактивном режиме — при этом пользователь должен подтверждать каждый свой шаг, что вынуждает задумываться, прежде чем что-то сделать. Имейте в виду, что некоторые исправления могут привести к определенным потерям данных, что отображается в диагностических сообщениях.

В интерактивном режиме перед каждым исправлением программа `fsck` ожидает ответа от пользователя — "Да" (`YES`) или "Нет" (`NO`). Если при запуске утилиты указать параметр `-y`, `fsck` допускает ответ "Да" и не делает паузы для ответа. Замечу, что программа `fsck` — многопроходная команда контроля файловых систем, поэтому на каждом проходе файловой системы выполняются различные этапы: проверка блоков и размеров файлов, полных имен файлов, связности, карты свободных блоков (возможно, с ее перестройкой), подсчет ссылок и т. д.

Перед запуском утилиты `fsck` файловая система должна быть демонтирована. Кроме того, если исправления вносятся в критическую файловую систему, например, в корневую, то после `fsck` систему необходимо перезагрузить.

5.3. Права доступа к файлам

Все объекты файловой системы имеют те или иные атрибуты доступа, позволяющие или запрещающие выполнять те или иные операции как отдельным пользователям, так и группам пользователей.

Для каждого файла устанавливается определенная комбинация прав доступа, представляющая собой набор из девяти битов. Она определяет, например, кто имеет право читать содержимое файла, записывать в него данные или выполнять его (если это исполняемый файл). Биты этого набора вместе с другими тремя битами, определяющими способ запуска исполняемых файлов, образуют код режима доступа к файлу. Все двенадцать битов режима хранятся в 16-битовом поле индексного дескриптора вместе с четырьмя дополнительными битами, указывающими тип файла.

Четыре последних бита устанавливаются в момент создания файла и не должны изменяться. Биты доступа может изменить либо владелец файла, либо суперпользователь root при помощи команды `chmod`. Эти биты вместе с остальной информацией отображаются на экране командой `ls`.

В коде режима доступа есть биты специального назначения, которым соответствуют восьмеричные значения 4000 и 2000. Это биты смены идентификатора пользователя (SUID) и смены идентификатора группы (SGID), позволяющие программам пользователя получать доступ к файлам и процессам, которые в иных обстоятельствах недоступны пользователю.

Бит, которому в коде режима доступа соответствует восьмеричное значение 1000, называется sticky-битом. В ранних версиях UNIX с небольшим объемом памяти, когда требовалось, чтобы отдельные программы постоянно оставались в памяти, sticky-бит был очень важен — он запрещал выгрузку программ из памяти. В настоящее время, когда повсеместно применяются недорогие и емкие модули памяти и быстродействующие дисковые накопители, sticky-бит уже не нужен, поэтому современные UNIX-системы его игнорируют.

Особенностью UNIX является то, что нельзя устанавливать биты прав доступа отдельно для каждого пользователя. Вместо этого используются разные трехбитовые комбинации (триады) для владельца файла, группы, которой принадлежит файл, и прочих пользователей. Каждая триада состоит из бита чтения, бита записи и бита выполнения (для каталога последний называется битом поиска).

Код режима доступа часто представляют в виде восьмеричного числа, т. к. каждая цифра в нем выражается тремя битами со следующими значениями:

- три старших бита (восьмеричные значения 400, 200 и 100) определяют права доступа к файлу со стороны его владельца;
- три средних бита (восьмеричные значения 40, 20 и 10) задают доступ для пользователей группы;
- младшие три бита (восьмеричные значения 4, 2 и 1) определяют права доступа к файлу для остальных пользователей.

При этом старший бит каждой триады определяет доступ по чтению, средний — доступ по записи и, наконец, младший бит определяет права на выполнение.

Каждый пользователь включается только в одну из категорий, соответствующих одной из триад битов режима, при этом из получившихся комбинаций выбираются те, которые гарантируют самые строгие права доступа. Так, например, права доступа для владельца файла всегда определяются триадой битов владельца и никогда — битами группы. При этом вполне возможна ситуация, когда другие пользователи наделены бóльшими правами доступа к файлу, чем владелец, но подобные конфигурации используются довольно редко.

Для текстового файла, например, установленный бит чтения разрешает читать данные, а бит записи — изменять данные. Удалить или переименовать файл можно в том случае, если установлены соответствующие биты прав доступа для каталога, где хранится имя файла.

Бит выполнения определяет возможность выполнить файл как программу или командный сценарий. Программа представляет собой исполняемый файл, содержащий двоичные данные, интерпретируемые как команды процессора и выполняемые непосредственно центральным процессором. Командный сценарий или командный файл обрабатывается интерпретатором shell или какой-нибудь другой программой (например, sed или Perl).

Для каталогов интерпретация бита выполнения несколько иная, чем для файла. Если для каталога установлен только бит выполнения, то разрешается вход в него, но получить список его содержимого нельзя. Содержимое каталога можно просмотреть в том случае, если установлены биты чтения и выполнения. Установленные биты записи и выполнения позволяют создавать, удалять и переименовывать файлы в данном каталоге.

Права доступа к файлам и каталогам можно установить при помощи команд `chmod` и `chown`, причем выполнять подобные действия может либо владелец файла, либо суперпользователь root.

Напомню, что команда `chmod` устанавливает права доступа к указанным файлам и имеет следующий синтаксис:

```
chmod [-fR] абсолютные_права файл ...
chmod [-fR] символьные_права файл ...
```

Первым аргументом команды `chmod` является спецификация прав доступа. Второй и последующий аргументы задают имена файлов, а также права доступа, подлежащие изменению. Изначально код доступа в операционных системах UNIX задавался в виде восьмеричного числа, хотя в современных версиях поддерживается и более наглядная система мнемонических обозначений. Восьмеричное представление более удобно для системных администраторов, хотя при этом можно задать только абсолютное значение режима доступа. Преимуществом мнемонического обозначения прав доступа является то, что можно сбрасывать и устанавливать отдельные биты режима.

При использовании восьмеричной нотации первая цифра относится к владельцу, вторая — к группе, а третья — к остальным пользователям. Если необходимо задать биты SUID/SGID или sticky-бит, следует указывать не три, а четыре восьмеричные цифры. Первая цифра в этом случае будет соответствовать трем специальным битам.

Опция `-f` команды отключает вывод сообщения о невозможности установки прав доступа. При помощи опции `-R` можно установить или изменить права доступа рекурсивно для всех подкаталогов, указанных в списке файлов.

Абсолютные права доступа, указанные опцией *абсолютные_права*, задаются восьмеричным числом, в то время как символьные права доступа, опреде-

ляемые опцией *символьные_права*, указываются в виде списка выражений (через запятую), например:

```
[пользователи] оператор [права, ...]
```

Элемент списка *пользователи* определяет, для кого задаются или изменяются права. Он может принимать значения `u`, `g`, `o` и `a`, относящиеся к владельцу, группе, остальным пользователям, а также ко всем категориям пользователей соответственно. Более подробно скажем, что символ `u` (user) обозначает владельца файла, символ `g` (group) — группу, символ `o` (others) — других пользователей, символ `a` (all) — всех пользователей сразу.

Если элемент не указан, права изменяются для всех категорий пользователей. Следует иметь в виду, что при этом не переопределяются установки, задаваемые маской создания файлов.

Элемент списка *оператор* может принимать значения `+`, `-` или `=`, означающие добавление, отмену права доступа и установку указанных прав соответственно. При этом если после оператора `=` ничего не указано, то все права доступа для соответствующих категорий пользователей отменяются.

Компонент *права* задается в виде любой совместимой комбинации следующих символов: `g`, `s`, `t`, `u`, `x`.

Не все сочетания символов для компонентов *пользователи* и *права* допустимы. Так, `s` можно задавать только для `u` или `g`, а `t` — только для `u`. Права `x` и `s` не совместимы с `t` и т. д. Изменения прав доступа в списке выполняются последовательно, в порядке их перечисления.

В табл. 5.1 показано восемь возможных комбинаций для каждого трехбитового набора, где символы `r`, `w` и `x` обозначают, соответственно, чтение, запись и выполнение.

Таблица 5.1. *Коды прав доступа в команде* `chmod`

Восьмеричное число	Двоичное число	Маска режима доступа
0	000	
1	001	`--x`
2	010	`-w-`
3	011	`-wx`
4	100	`r--`
5	101	`r-x`
6	110	`rw-`
7	111	`rwx`

Например, команда

```
chmod 711 myfile
```

предоставляет владельцу все права доступа к файлу myfile и право на выполнение — остальным пользователям. Мнемонические спецификации представлены в табл. 5.2.

Таблица 5.2. *Примеры мнемонических спецификаций команды* `chmod`

Спецификация	Описание
`u+w`	Владельцу файла дополнительно разрешается выполнять файл
`ug=rw,o=r`	Владелец и группа получают право на чтение/запись, остальные пользователи — право чтения
`a-x`	Все пользователи лишаются права выполнения
`ug=srx,o=`	Владелец и группа получают право чтения/выполнения, и устанавливается бит SUID; доступ к файлу остальным пользователям запрещен
`g=u`	Группе назначаются такие же права, что и владельцу

Рассмотрим пример изменения прав доступа. Предположим, что в текущем каталоге имеется файл test, для которого мы будем изменять права доступа, как показано далее, одновременно наблюдая за результатами:

```
$ chmod +w test
$ ls -l test
-rw-r--r--   1 yury 50            0 Dec 11 2:05 test
$ chmod a+w test
$ ls -l test
-rw-rw-rw-   1 yury 50            0 Dec 11 2:10 test
$ chmod u+x,g=x,o= test
$ ls -l test
-rwx--x---   1 yury 50            0 Dec 11 2:13 test
$ chmod ug-x,og+r,u=rwx test
$ ls -l test
-rwxr--r--   1 yury 50            0 Dec 11 2:14 test
$ chmod 644 test
$ ls -l test
-rw-r--r--   1 yury 50            0 Dec 11 2:15 test
```

При создании нового файла ему присваиваются права доступа, определяемые пользовательской маской режима создания файлов. Встроенная команда

umask интерпретатора shell присваивает пользовательской маске режима создания файлов указанное восьмеричное значение. Три восьмеричные цифры соответствуют правам на чтение/запись/выполнение для владельца, членов группы и прочих пользователей соответственно.

Команда имеет следующий синтаксис:

```
umask [-S] [маска]
```

Выполненная без параметров, команда umask отображает текущее значение маски. По умолчанию значение маски задается и отображается в восьмеричном виде как число. Маска используется при указании прав доступа следующим образом: это значение необходимо "вычесть" из максимальных прав доступа (777 для выполняемых файлов, которые создаются компиляторами, и 666 для обычных файлов). Вот пример:

```
$ umask
022
```

При данном значении маски обычные текстовые файлы будут создаваться с правами 666 – 022 = 644, как это показано на примере создания файла test:

```
$ > test
$ ls -l test
-rw-r--r--   1 yury 50             0 Nov 12 1:39 test
```

Операция "вычитания" для значения маски формально выполняется как побитовое логическое И дополнения маски и максимальных прав доступа. Для нашего примера расчет прав доступа посредством маски показан в табл. 5.3.

Таблица 5.3. *Расчет маски*

Значение	Результат
Двоичное значение маски	000010010 (022)
Дополнение маски	111101101 (755)
Максимальное значение прав	110110110 (666)
Логическое И предыдущих двух строк	110100100 (644)
Результирующие биты прав	110100100 (644)

Опция -S требует отображать значение маски в символьном виде. В этом случае отображаются биты прав доступа у вновь создаваемого файла. При этом берется во внимание и бит выполнения:

```
$ umask -S
u=rwx,g=rx,o=rx
```

Команду `umask` нередко включают в файлы начального запуска, устанавливающие рабочую среду для начального командного интерпретатора.

Вот еще один пример создания файла, но при другом значении маски (257):

```
$ umask 257
$ > test
$ ls -l test
-r---w----   1 yury 50             0 Nov 12 10:34 test
```

Владелец файла, а также суперпользователь root могут изменить владельца и группу-владельца файла, используя команду `chown`.

Команда имеет такой синтаксис:

```
chown [-h][-R] владелец[:группа] файл ...
```

Если нужно изменить только группу, владеющую файлом, можно использовать команду `chgrp`:

```
chgrp [-h][-R] группа файл
```

В качестве первого аргумента обеих команд используется имя нового владельца или новой группы соответственно. Кроме того, для выполнения команды `chgrp` необходимо быть владельцем файла и входить в назначаемую группу, или иметь права суперпользователя root.

Опция `-h` требует изменить владельца файла, на который указывает символическая ссылка, а не саму ссылку, как это принято по умолчанию.

В большинстве версий команд `chown` и `chgrp` предусмотрен флаг `-R`, который задает смену владельца или группы не только самого каталога, но и всех его подкаталогов и файлов. Например, последовательность команд

```
chmod 755 -red
chown -R red -red
chgrp -R color -red
```

можно использовать для конфигурирования начального каталога нового пользователя после копирования в него стандартных файлов сценариев. Не следует пытаться выполнять команду `chown` для файлов, имена которых начинаются с точки:

```
#chown  -R red -red/.*
```

Указанному шаблону поиска соответствует также файл ~red/.., вследствие чего команда изменит и владельца родительского каталога.

В операционных системах UNIX, совместимых с System V, пользователи могут беспрепятственно поменять владельца собственного файла при помощи

команды chown, тогда как в BSD-совместимых системах эту команду может выполнять только суперпользователь root.

Необходимо учитывать, что после смены владельца файла бывший владелец будет иметь права доступа, установленные новым владельцем. Пусть требуется сменить владельца файла test, атрибуты которого показаны далее:

```
$ ls -l
total 2
-rw-r--r--    1 user1   others        6 Dec 10 16:19 test
```

Из результата выполнения команды ls видно, что владельцем файла test является пользователь user1, в сеансе которого и выполняются команды. Сменим владельца файла на yury:

```
$ chown yury test
$ ls -l
total 2
-rw-r--r--    1 yury others          6 Dec 10 16:19 test
```

После выполнения команды chown видно, что пользователем файла test является yury. Если теперь в сеансе пользователя user1 попробовать заменить владельца опять на user1, то будет выдана ошибка:

```
$ chown user1 test
UX:chown: ERROR: testf: Not privileged
```

Помимо рассмотренных нами атрибутов доступа, объекты файловой системы операционной системы UNIX имеют и другие атрибуты. Рассмотрим их более подробно. Напомню, что информация о каждом файле хранится в структуре, называемой индексным дескриптором.

Индексный дескриптор содержит порядка сорока информационных полей, но большинство из них используется только ядром. Пользователя операционной системы интересует, как правило, только небольшая часть информации, находящейся в индексных дескрипторах:

- количество жестких ссылок;
- владелец файла;
- группа-владелец файла;
- код прав доступа;
- размер файла;
- время последнего обращения;
- время последней модификации;
- тип файла.

Всю эту информацию можно легко получить с помощью команды `ls -l`. Рассмотрим пример:

```
$ ls -l /bin/sh
-rwxr-xr-x  1   root  bin  85924 Sep 27  1997 /bin/sh
```

Здесь в первом поле задаются тип файла и маска режима доступа к нему. Первый символ — дефис, следовательно, это обычный файл. Обозначения различных типов файлов представлены односимвольными кодами (табл. 5.4).

Таблица 5.4. *Кодирование типов файлов в команде* `ls -l`

Тип файла	Символ	Создается командой	Удаляется командой
Обычный файл	-	Редакторы, `cp` и др.	—
Каталог	`d`	`mkdir`	`rmdir`, `rm -r`
Файл байт-ориентированного устройства	`c`	`mknod`	`rm`

Следующие девять символов в этом поле представляют три набора (триады) битов режима. В выводе команды `ls` они представляются литерами `r`, `w` и `x` (чтение, запись и выполнение соответственно). В данном примере владелец обладает полным доступом к файлу, а остальные пользователи — только правом на чтение и выполнение.

Если бы был установлен бит смены идентификатора пользователя (SUID), то вместо `x` стоял бы символ `s`. Аналогично, если бы был установлен бит смены идентификатора группы (SGID), то вместо `x` для группы стоял бы символ `s`. Последний бит режима (право выполнения для остальных пользователей) представляется буквой `t`, когда для файла задан sticky-бит. Если биты SUID/SGID или sticky-бит установлены, а надлежащий бит выполнения — нет, эти биты представляются, соответственно, символами, указывающими на наличие ошибки и игнорирование данных атрибутов.

Следующее поле содержит счетчик ссылок на файл. На этом месте стоит единица, а это значит, что /bin/sh — единственное имя, под которым известен данный файл. Всякий раз при создании жесткой ссылки на файл этот счетчик увеличивается на единицу. Что же касается каталогов, то любой из них имеет как минимум две жесткие ссылки: одну из родительского каталога и одну из специального файла внутри самого каталога. Следует отметить, что символические ссылки в счетчике не учитываются.

Следующие два поля — владелец и группа-владелец файла. В данном случае владельцем файла является суперпользователь root, а файл принадлежит

группе bin. Ядро операционной системы хранит эти данные не в виде строк, а как идентификаторы пользователя и группы. Если невозможно получить символьное представление имен, в этих полях будут отображаться числа. Такое случается, если запись пользователя или группы была удалена из файла /etc/passwd или /etc/group соответственно.

Далее следует поле, отображающее размер файла в байтах: данный файл имеет размер 85 924 байта, т. е. почти 84 Кбайт. Следующее поле содержит дату последнего изменения: 27 сентября 1997 г., и, наконец, в последнем поле вывода содержится имя файла: /bin/sh.

Если мы имеем дело с файлом устройства, то вывод команды `ls` будет другим, например:

```
$ ls -l /dev/ttya
crw-rw-rw-  1   root  daemon     12,   0  Dec  20   1998   /dev/ttya
```

Результат работы этой команды иной, чем в предыдущем примере: вместо размера в байтах показаны старший и младший номера устройства. Имя /dev/ttya относится к первому устройству, управляемому драйвером устройства 12 (в данной системе это драйвер терминала).

При поиске жестких ссылок часто бывает полезной команда `ls -i`, отображающая для каждого файла номер индексного дескриптора. Жесткие ссылки, указывающие на один и тот же файл, будут иметь один и тот же номер.

Операционная система автоматически отслеживает такие атрибуты, как время изменения, число ссылок и размер файла, автоматически устанавливая корректные значения. В то же время права доступа и идентификаторы принадлежности файла могут быть модифицированы явным образом с помощью команд `chmod`, `chown` и `chgrp`.

5.4. Операции с файлами

Значительная часть времени пользователя уходит на операции с файлами, такие как копирование, перемещение, удаление и поиск. Далее мы проанализируем, какие средства предлагает операционная система UNIX для выполнения таких задач. Начнем с копирования и перемещения файлов.

5.4.1. Копирование файлов

Для копирования объектов файловой системы используется команда `cp`. С ее помощью копируются файлы или каталоги, причем можно либо копировать

один файл в другой, заданный файл, либо копировать группу файлов в заданный каталог. Синтаксис команды таков:

```
cp [опции] файл путь
cp [опции] файл... каталог
```

Остановимся более детально на специфике работы команды `cp`. Если последним параметром является существующий каталог, то `cp` копирует исходные файлы в этот каталог, сохраняя при этом их имена. Если в качестве параметров заданы два файла, то `cp` копирует первый файл во второй.

В том случае, если командная строка содержит более двух параметров, не являющихся опциями самой команды, а последний параметр не является именем какого-либо каталога, то возникает ошибка.

Например, если /a — каталог, то команда

```
cp -r /a /b
```

скопирует /a в /b/a и /a/x в /b/a/x в том случае, если каталог /b уже существует. Если /b не существует, то эта команда создаст его и скопирует /a в /b и /a/x в /b/x. Наконец, если /b является обычным файлом, то команда завершится с ошибкой.

Права доступа к скопированным файлам и каталогам вычисляются путем логического умножения (операция И) кода доступа исходных файлов на 0777, а также с учетом `umask` пользователя (за исключением случая, когда задана опция `-p`). Если выполняется рекурсивное копирование каталогов, то вновь создаваемые каталоги временно получают права доступа, вычисленные путем логического сложения (операция ИЛИ) исходного кода доступа со значением `S_IRWXU` (0700), разрешая тем самым чтение, запись и поиск во вновь созданных каталогах.

Попытка скопировать файл сам в себя ни к чему не приводит, кроме того, что выдается сообщение об ошибке.

5.4.2. Удаление файлов

Для удаления файла с помощью команды `cp` необходимо указать опцию `-f`.

Команда поддерживает как опции стандарта POSIX, так и опции GNU. Вот опции POSIX:

- `-f` — удаляет существующие файлы;
- `-i` — выдает запрос относительно перезаписи существующих файлов, в которые происходит копирование. Копирование выполняется только в случае положительного ответа;

- `-p` — сохраняет исходные атрибуты файла, такие как владелец, группа, права доступа (включая SUID- и SGID-биты), время последней модификации и время последнего доступа к файлу. Если попытка установки владельца или группы приводит к ошибке, биты SUID и SGID сбрасываются;
- `-R` — выполняет рекурсивное копирование каталогов, отслеживая ситуации, когда встречаются объекты, не являющиеся обычными файлами или каталогами, например, именованные каналы (FIFO) или специальные файлы. В таких случаях создаются корректные копии таких объектов;
- `-r` — выполняет рекурсивное копирование каталогов, производя при этом не определенные стандартом действия, если встречаются объекты, не являющиеся обычными файлами или каталогами.

В стандарт POSIX 1003.1-2003 добавлены три опции, определяющие способы обработки символических ссылок:

- при выполнении нерекурсивного копирования символические ссылки разыменовываются;
- при выполнении рекурсивного копирования с опцией `-r` результаты зависят от реализации;
- при выполнении рекурсивного копирования с опцией `-R` необходимо принимать во внимание дополнительные опции:
  - `-H` — разыменовываются символические ссылки, указанные в списке параметров. В то же время символические ссылки, которые встречаются во время рекурсивного копирования, копируются;
  - `-L` — разыменовываются все символические ссылки, как заданные в списке параметров, так и встретившиеся при выполнении рекурсивного копирования;
  - `-P` — не разыменовываются никакие символические ссылки, как те, что заданы в списке параметров, так и те, которые встречаются во время рекурсивного копирования. В этом случае выполняется обычное копирование символических ссылок.

Во всех последующих примерах применения команды `cp` мы будем придерживаться стандарта POSIX.

Вот некоторые примеры использования команды `cp`. Для копирования одного каталога в другой можно выполнить команду:

```
$ cp -r DIR ARCH_DIR
```

Здесь каталог DIR вместе со своим содержимым копируется в каталог ARCH_DIR.

В следующем примере команда

```
$ cp -r USR1 USR2 ARCH
```

копирует содержимое каталогов USR1 и USR2 в каталог ARCH.

5.4.3. Перемещение файлов

Остановимся на операции перемещения файлов. Перемещение файлов в операционной системе UNIX выполняется с помощью команды `mv`, имеющей следующий синтаксис:

`mv [`*опции*`...] `*исходный_файл файл_назначения*

`mv [`*опции*`...] `*исходный_файл*`... `*каталог*

Если последний аргумент является именем существующего каталога, то `mv` перемещает указанные файлы в этот каталог. В том случае, если заданы только два файла, то имя первого файла будет изменено на имя второго. Наконец, если последний аргумент не является каталогом, и задано более чем два файла, то будет выдано сообщение об ошибке.

Так, команда

```
mv /a/x/y /b
```

переименует файл /a/x/y в /b/y, если /b является существующим каталогом, и в /b, если нет.

Если при переименовании *исходного_файла* в *файл_назначения* последний существует (при заданной опции `-i`) или если выполнить запись в него невозможно, а стандартным выводом является терминал, и не задана опция `-f`, то `mv` запрашивает у пользователя разрешение на замену этого файла. Если ответ отрицательный, то файл пропускается.

Когда *исходный_файл* и *файл_назначения* находятся в одной файловой системе, то изменяется имя файла, а владелец, права доступа, временные штампы остаются неизменными. Если же они находятся в разных файловых системах, то *исходный_файл* копируется и затем удаляется. Во время выполнения операции команда `mv`, если это возможно, будет копировать время последней модификации, время доступа, идентификаторы пользователя и группы и права доступа к файлу. Если копирование идентификаторов пользователя и/или группы закончилось неудачно, то биты SETUID и SETGID скопированного файла сбрасываются.

Поддерживаются следующие опции POSIX:

- `-f` — запрос на подтверждение операции не выдается;

- `-i` — выдается запрос на подтверждение операции, когда *`файл_назначения`* существует. Если заданы обе опции `-f` и `-i`, используется только последняя из них.

Вот примеры использования команды `mv`:

```
$ mv qt qt.OLD
```

Здесь файл test переименовывается в файл test.old.

К недостаткам команды `mv` следует отнести то, что она не работает с шаблонами файлов приемника и источника. Если бы потребовалось, например, чтобы все файлы, соответствующие шаблону `t*`, были переименованы в файлы t*.old, то возникла бы ошибка:

```
$ mv t* t*.old
mv: when moving multiple files, last argument must be a directory
Try 'mv --help' for more information.
```

5.4.4. Создание каталогов

Команды, которые мы рассматривали до сих пор, в основном работают с обычными файлами. Для управления каталогами в системе UNIX имеются две команды — `mkdir` и `rmdir`.

Новый каталог можно создать с помощью команды `mkdir`. В самой простой форме эта команда использует один параметр — имя каталога — и создает каталог с этим именем.

Опытные пользователи с помощью одной команды `mkdir` могут создавать несколько каталогов сразу, перечисляя их в одной командной строке. Синтаксис команды таков:

```
mkdir [опции] [список_каталогов]
```

Вместе с командой `mkdir` можно использовать две опции. Опция `-m` позволяет задать в восьмеричной или символьной форме права доступа (как и для команды `chmod`), которые будут присвоены создаваемым каталогам. При использовании ключа `-p`, кроме указанного каталога, будут созданы также и любые требуемые промежуточные каталоги. При этом если у пользователя нет прав на запись в родительский каталог, то новый каталог не будет создан. Если каталог уже существует или вместо каталога существует файл с таким же именем, то будет выведено сообщение об ошибке.

5.4.5. Удаление каталогов

Для удаления файлов и каталогов в операционной системе UNIX используются команды `rm` и `rmdir`. С помощью команды `rmdir` проще удалить оди-

ночный каталог, причем он должен быть пустым. Если в каталоге имеются элементы, отличные от . и .., то команда `rmdir` такой каталог не удаляет. Синтаксис этой команды таков:

```
rmdir [-p][-s] каталог
```

Команда `rmdir` имеет две опции:

- `-p` — позволяет удалить пустой каталог вместе с его родительскими каталогами, отображая сообщение об успешном или неуспешном выполнении операции;
- `-s` — подавляет выдачу сообщений при использовании опции `-p`.

Команда `rm` работает несколько иначе. С ее помощью можно удалить указанные файлы, при этом каталоги по умолчанию не удаляются.

Если для команды заданы опции `-r` или `-R`, то будет удаляться все дерево каталогов ниже заданного каталога, включая и сам каталог, причем не накладывается никаких ограничений на глубину дерева. Если последний компонент файла — это . или .., то выдается сообщение об ошибке (это сделано для того, чтобы избежать неприятных сюрпризов при выполнении команды `rm -r .*` или ей подобных). Команда `rm` поддерживает стандарты POSIX и GNU.

Вот опции POSIX:

- `-f` — не запрашивается подтверждение операции и не выдаются диагностические сообщения. При завершении команды с ошибками код ошибки не возвращается, если ошибки вызваны отсутствием файлов;
- `-i` — выводится запрос на подтверждение удаления (при указании опций `-f` и `-i` одновременно используется последняя);
- `-r` или `-R` — позволяет рекурсивно удалять дерево каталогов.

Следует учитывать потенциальную опасность использования команды `rm`, поскольку после ее выполнения восстановить удаленные файлы невозможно. Вот некоторые примеры неправильного использования команды `rm`.

Например, командная строка

```
rm * /tmp
```

должна была по замыслу удалить все файлы в каталоге /tmp. На самом деле были удалены сначала все файлы в текущем каталоге, а затем осуществлена попытка удалить /tmp, которая закончилась неудачно, поскольку tmp — это каталог.

Правильно следовало бы ввести команду

```
rm /tmp/*
```

5.4.6. Поиск файлов и каталогов

Поиск файлов и каталогов также относится к довольно часто выполняемым операциям. Поиск файлов и каталогов, удовлетворяющих определенным критериям, как известно, выполняется стандартными средствами операционной системы. Файловая система UNIX содержит тысячи файлов, поэтому для быстрого поиска нужны очень эффективные средства, одним из которых является команда `find` с соответствующими опциями, представляющая собой очень мощный инструмент для выполнения подобных операций.

Команда имеет следующий синтаксис:

`find` *каталог* `...` *выражение*

Утилита просматривает иерархии каталогов в поисках файлов, удовлетворяющих критерию, задаваемому выражением *выражение*. Выражения строятся из элементов с помощью следующих конструкций:

- `-name` *шаблон* — условие истинно, если имя файла соответствует шаблону. При использовании метасимволов необходимо маскировать шаблоны от командного интерпретатора;
- `-type` *тип* — условие истинно, если файл — указанного типа. Типы файлов задаются символами `b`, `c`, `d`, `f`, `l`, `p` и `s`, обозначающими, соответственно, специальное блочное устройство, специальное символьное устройство, каталог, обычный файл, символическую ссылку, именованный канал и сокет;
- `-user` *пользователь* — условие истинно, если файл принадлежит пользователю, указанному по идентификатору или регистрационному имени;
- `-group` *группа* — условие истинно, если файл принадлежит группе, указанной по идентификатору или имени;
- `-perm [-]` *права* — если дефис не задан, то условие истинно, только если права доступа в точности соответствуют указанным (как в команде `chmod`). Если задан дефис, то условие истинно, если в правах доступа файла, как минимум, установлены те же биты, что и в указанных правах;
- `-size [+|-|=]n[c]` — условие истинно, если файл имеет длину `n` блоков (блок — 512 байтов) или символов (если указан суффикс `c`). Перед размером можно указывать префикс `+` (не меньше), `-` (не больше) или `=` (в точности равен);
- `-atime [+|-|=]n` — условие истинно, если к файлу последний раз обращались `n` дней назад. Перед `n` в элементах `-atime`, `-ctime` и `-mtime` можно указывать префикс `+` (не позже), `-` (не ранее) или `=` (ровно);
- `-ctime n` — условие истинно, если файл создан `n` дней назад;

- `-mtime n` — условие истинно, если файл был изменен `n` дней назад;
- `-newer` *файл* — условие истинно, если файл — более новый, чем указанный;
- `-ls` — условие истинно всегда (выдает информацию о файле, аналогичную длинному листингу);
- `-print` — условие истинно всегда (выдает полное имя файла в стандартный выходной поток);
- `-exec` *команда* `{} \;` — условие истинно, если выполненная команда имеет код возврата 0. Команда заканчивается замаскированной точкой с запятой. В команде можно использовать конструкцию `{}`, заменяемую полным именем рассматриваемого файла;
- `-ok` *команда* `{} \;` — аналогично `exec`, но полученная после подстановки имени файла вместо `{}` команда выдается с вопросительным знаком и выполняется, если пользователь ввел символ `y`;
- `-depth` — условие истинно всегда — требует так обходить иерархию каталогов, чтобы файлы любого каталога всегда обрабатывались раньше, чем сам каталог (обход "в глубину");
- `-prune` — условие истинно всегда — требует не проверять файлы в каталоге, путевое имя которого присутствует в предыдущем выражении. Не действует, если ранее указан элемент `-depth`.

В различных версиях операционной системы UNIX могут поддерживаться и другие компоненты выражений в команде `find`. Следует сказать, что если командная строка сформирована неправильно, команда немедленно завершает работу.

Рассмотрим несколько примеров использования команды `find`.

Вначале — простой пример поиска файлов в каком-либо каталоге и вывод содержимого на консоль. Например, чтобы получить содержимое рабочего каталога программы (в нашем примере это /home/yury), достаточно выполнить команду

```
$ find `pwd` -print
```

На экране дисплея будет отображено содержимое текущего каталога:

```
/home/yury
/home/yury/developer
/home/yury/developer/test.c
```

Для получения содержимого другого каталога, например, /home/yury/developer, нужно выполнить команду

```
$ find /home/yury/developer -print
/home/yury/developer
```

```
/home/yury/developer/MISC
/home/yury/developer/MISC/test
/home/yury/developer/MISC/misc.c
/home/yury/developer/MISC/stat_name.c
```

Следующая команда выводит на консоль содержимое текущего каталога, откуда была запущена команда `find`:

```
$ find . -print
.
./developer
./developer/test.c
```

Вызванная с параметрами, указанными далее, команда `find` ничего не выдает, поскольку файла с именем tmp в текущем каталоге нет:

```
$ find . -name tmp -print
```

Следующий пример отличается от предыдущего шаблоном имени файла. Пусть необходимо найти файлы, чьи имена заканчиваются на tmp. Если текущий каталог содержит, например, такие файлы:

```
$ ls -l
total 3
-rw-r--r--    1 YURY     Отсутств        5   Feb  3 18:08 test1
-rw-r--r--    1 YURY     Отсутств        199 Feb  3 18:08 test1.c
-rw-r--r--    1 YURY     Отсутств        9   Feb  3 18:09 test3.tmp
```

то поиск легко выполнить при помощи командной строки, указанной далее (здесь же выводится и результат поиска):

```
$ find . -name '*tmp' -print
./test3.tmp
```

В следующем примере с помощью команды `find` выполняется поиск файлов с расширением tmp или c, находящихся в текущем каталоге:

```
$ find . \( -name '*.tmp' -o -name '*.c' \) -print
./test1.c
./test3.tmp
```

В команде `find` можно задавать временны́е критерии поиска файлов, причем в самых различных комбинациях. Следующий пример демонстрирует это: в нем используется опция `-atime [+|-|=]n`. Условие является истинным, если время последнего доступа к файлу больше/меньше, чем `n*24`. Например, команда

```
$ find . \( -name '*.tmp' -o -name '*.pl' \) -atime +3 -print
```

выполняет поиск файлов с указанными шаблонами, к которым не было обращения больше трех суток.

Нередко требуется найти файлы, принадлежащие определенному пользователю. Например, следующая команда выполняет поиск файлов в каталоге DIR, владельцем которых является суперпользователь root:

```
$ find DIR -user root -print
```

Если критерием поиска является размер файла, то можно использовать следующую опцию: `-size [+|-|=]n[c]`. Условие, задаваемое этой опцией, истинно, если размер файла больше/меньше `n`. При этом различают два случая: если присутствует опция `c`, то размер файла предполагается заданным в байтах, если опция `c` отсутствует — то в блоках по 512 байтов.

Следующая команда выполняет поиск файлов, размер которых превышает 256 байтов, в каталоге DIR:

```
$ find DIR -size +256c -print
```

Особую гибкость команде `find` дает еще одна возможность — выполнение команды или группы команд, принимающих в качестве параметра результат поиска файлов. Для реализации такой возможности служит опция `-exec`. В этом случае команда должна заканчиваться пробелом и символами `\;`.

Рассмотрим следующий пример. Пусть в каталоге DIR необходимо удалить все файлы, размер которых не превышает 100 байтов. Подобную операцию можно выполнить при помощи командной строки

```
$ find DIR -size -100c -print -exec rm {} \;
```

Приведу еще один пример использования опции `-exec`:

```
$ find TEST -name 't*' -exec ls -l {} \;
```

Здесь на консоль выводятся атрибуты всех файлов (команда `ls -l`), удовлетворяющих шаблону `t*`.

Можно расширить возможности команды `find`, перенаправив ее вывод не на стандартное устройство вывода, а в программный канал. В этом случае значительно расширяются возможности поиска и обработки файлов.

Следующая команда

```
$ find DIR -name 't*' -print|grep tmp
```

выполняет поиск файлов в каталоге DIR, удовлетворяющих шаблону `t*`, в имени которых присутствует tmp.

Очень часто конвейер программ применяется в операциях копирования, перемещения и создания резервных копий файловых систем, когда вывод команды `find` служит вводом для команды `cpio`.

Команда `find` часто используется в командных файлах, когда нужно выполнить определенные однотипные действия над объектами файловой системы, удовлетворяющими определенным критериям.

Например, можно разработать командные файлы, позволяющие убрать мусор — ставшие ненужными временные файлы или файлы, имеющие нулевой размер. Такие файлы могут создаваться как самой операционной системой, так и работающими приложениями. Накапливаясь, такие файлы уменьшают свободное дисковое пространство и замедляют работу файловой системы в целом. Хочу сказать, что часть файлов, имеющих нулевой размер, является файлами устройств или системными, поэтому нужно осторожно подходить к каким-либо манипуляциям с такими файлами.

Рассмотрим несколько примеров командных файлов с использованием команды `find`. Следующий командный файл (назовем его find_zero) обнаруживает в текущем каталоге файлы, имеющие нулевой размер, и удаляет их:

```
if [ $# -ne 1 ]
 then
  echo "Usage: $0 [file]"
 else
  dir=`pwd`
  list=`find "$dir" -name "$1"`
  echo Searching zero-length files in current directory...
  for file in $list
   do
    if [ -f $file -a  ! -s $file ]
     then
      echo Found "$file"
      rm $file
      echo Successfully deleted.
    fi
  done
fi
echo Done.
```

Здесь переменной `dir` присваивается строка, представляющая путь к текущему каталогу, в котором выполняется командный файл:

```
dir=`pwd`
```

В качестве единственного параметра программа принимает шаблон имени файла `$1`. Для поиска файлов здесь используется команда `find`. Оператор

```
if [ -f $file -a  ! -s $file ]
```

выполняет проверку найденного объекта на принадлежность к файлам (опция `-f`) и на равенство его размера нулю (опция `! -s`).

Если задать командную строку, например, в виде

```
$ ./find_zero "*"
```

то будет выполнен поиск и удаление всех файлов нулевого размера.

Если задать командную строку в таком виде

```
$ ./find_zero "t*"
```

то ищутся и удаляются только файлы, начинающиеся с символа `t`.

5.5. Архивирование данных

В процессе функционирования в операционной системе UNIX происходит постоянное увеличение как количества файлов, так и их размера. Это рано или поздно приводит к нескольким неприятным вещам. Во-первых, заполняется дисковое пространство, и какова бы ни была емкость дисков, наступает момент времени, когда ее оказывается недостаточно. Во-вторых, замедляется работа самой операционной системы: чем больше файлов, тем больше времени требуется для выполнения файловых операций, что связано с произвольным расположением кластеров на диске. Как показывают наблюдения, доступ к большинству файлов происходит довольно редко, особенно это касается программ, управляющих базами данных. В таких случаях имеет смысл хранить редко используемые файлы в виде архивов. Еще один важный момент — архивы данных нужны при авариях системы, когда требуется восстановить данные за определенный период времени.

Для операций архивирования и обратных им операций восстановления в операционной системе UNIX предусмотрен целый ряд команд, в их числе и уже знакомая нам команда `find`.

Команда `find` очень широко используется при операциях архивирования и перемещения файлов и целых файловых систем. Чаще всего в таких операциях она применяется в паре с командой `cpio`, позволяющей копировать файлы в архив или извлекать их из архива. Опции команды `-i`, `-o` и `-p` задают требуемую операцию.

Команда `cpio -i` (от англ. *copy in*) извлекает файлы из стандартного входного потока, предположительно сформированного предыдущей командой `cpio -o`. Выбираются только имена файлов, соответствующие шаблону. Извлекаемые файлы копируются в текущее дерево каталогов в соответствии с дополнительными опциями. Права доступа для созданных файлов соответ-

ствуют тем, которые были во время создания архива командой `cpio -o`. При этом атрибуты пользователя и группы-владельца устанавливаются в соответствии с теми, которые имеются у текущего пользователя, если только это не суперпользователь root — в этом случае владельцы будут такие же, как и при выполнении соответствующей команды `cpio -o`.

Используемая с опцией `-i` команда имеет синтаксис:

```
cpio -i [ bBcdfkmPrsStuvV6 ] [ -C размер_буфера ]
        [ -E файл ] [ -H формат ] [ -I файл [ -M сообщение ] ]
        [ -R id ] [ шаблон ... ]
```

Смысл дополнительных опций (не только для `-i`) мы рассмотрим немного позже. Должен заметить, что при попытке создания файла, который уже существует, с той же датой изменения или более новой команда `cpio -i` выдает предупреждающее сообщение и не перезаписывает существующий файл. Если необходимо безусловно перезаписать существующие файлы, следует задать опцию `-u`.

Команда `cpio -o` (от англ. *copy out*) читает список файлов из входного потока и копирует эти файлы в стандартный выходной поток вместе с полным путевым именем и информацией о состоянии. Результат по умолчанию выравнивается по 8192-байтовой границе или до указанного пользователем (с помощью опций `-B` или `-C`) размера блока, или, при необходимости, до специфического размера блока устройства. Команда `cpio -o` имеет ряд дополнительных опций, представленных далее, смысл которых мы рассмотрим позже:

```
cpio -o [ aABcLPvV ] [ -C размер_буфера ]
        [ -H формат ] [ -O файл [ -M сообщение ] ]
```

Команда `cpio -o` использует стандартный входной поток, причем, если он формируется и направляется по программному каналу команде `cpio -o`, утилита группирует файлы так, что они могут быть перенаправлены (`>`) в один файл архива. Опция `-c` (как и опция `-H`) гарантирует, что файл архива будет переносим на другие машины. Вместо `ls` можно использовать `find`, `echo`, `cat` и любые другие команды, формирующие в стандартном выходном потоке список имен файлов для `cpio`. Результат можно перенаправить на устройство, а не в обычный файл.

Еще один режим работы команды `cpio` — режим передачи. В этом режиме `cpio -p` (от англ. *pass*) читает список файлов, которые при необходимости, в зависимости от описанных далее опций, создаются и копируются в указанное целевое дерево каталогов.

```
cpio -p [ adlLmPuvV ] [ -R id ] каталог
```

Проанализируем более детально смысл некоторых дополнительных опций, которые можно в любой последовательности указывать вместе с опциями `-o`, `-i` или `-p`:

- `-a` — выполняет сброс времени обращения к исходным файлам после их копирования. При этом время обращения не сбрасывается для связанных файлов при указании опций `cpio -pla`;
- `-A` — добавляет файлы в архив, требуя обязательного указания опции `-O`. Использование данной опции допустимо только для архивов в файлах, на дискетах или на разделах жесткого диска;
- `-b` — изменяет порядок байтов в каждом слове и используется исключительно с опцией `-i`;
- `-B` — осуществляет ввод/вывод в виде блоков по 5120 байтов. Если эта опция и опция `-C` не указаны, то используется стандартный размер блока — 8192 байта. Опцию `-B` нельзя использовать в режиме передачи; кроме того, она имеет смысл только при обмене данными со специальным символьным устройством, например, /dev/rmt/0m;
- `-C` *размер_буфера* — объединяет ввод/вывод в записи указанного размера (размер задается положительным целым числом). Стандартный размер буфера, если не задана эта опция и опция `-B`, равен 8192 байта. Опция `-C` имеет смысл, только если происходит обмен данными со специальным символьным устройством, например, /dev/rmt/0m;
- `-d` — создает каталоги при необходимости;
- `-E` *файл* — задает входной `файл`, содержащий список имен файлов, которые необходимо извлечь из архива (по одному имени в строке).

Если при записи на специальное символьное устройство (`-o`) или при чтении со специального символьного устройства (`-i`) утилита `cpio` обнаруживает конец носителя (например, конец дискеты), и при этом не заданы опции `-O` и `-I`, выдается такое сообщение:

```
To continue, type device/file name when ready.
(Для продолжения введите имя устройства/файла.)
```

Для продолжения необходимо заменить носитель, ввести имя специального символьного устройства (например, /dev/rdiskette) и нажать клавишу <Enter>. Можно для продолжения архивирования задать `cpio` для другого устройства. Например, при наличии двух дисководов можно переключать архивирование с одного на другой, чтобы оно продолжалось, пока вы меняете дискету. (Если просто нажать клавишу <Enter>, процесс `cpio` завершает работу.)

Кроме дополнительных опций команда `cpio` поддерживает следующие операнды:

- *`каталог`* — путь к существующему целевому каталогу для команды `cpio -p`;
- *`шаблон`* — выражение с теми же метасимволами сопоставления с образцом, что и шаблоны имен файлов командного интерпретатора:
  - `*` — соответствует любой строке, в том числе пустой;
  - `?` — соответствует любому одиночному символу;
  - `[...]` — соответствует любому из указанных в квадратных скобках символов. Пара символов, разделенных дефисом (`-`), соответствует любому из символов диапазона (включительно с указанными), определяемых по стандартной кодовой таблице. Если сразу за открывающей скобкой (`[`) идет восклицательный знак (`!`), то результаты будут неопределенными;
  - `!` — отрицание (например, `!abc*` исключает из обработки все файлы, имена которых начинаются с abc).

В шаблонах метасимволы `?`, `*` и `[...]` сопоставляются с символом косой черты (`/`), а символ обратной косой черты (`\`) является маскирующим. Можно задавать несколько шаблонов, а если шаблон не задан, предполагается шаблон `*` (т. е. выбираются все файлы). Каждый шаблон необходимо брать в двойные кавычки, в противном случае командный интерпретатор может подставить имена файлов текущего каталога.

Рассмотрим несколько примеров использования утилиты `cpio` для архивирования и восстановления файлов.

Пусть требуется создать архив файлов текущего каталога и записать его в файл arch. Предположим, что текущий каталог содержит такие файлы:

```
$ ls -l
-rw-r--r--    1 yury     yury       10 Feb  5 17:32 memo1
-rw-r--r--    1 yury     yury       14 Feb  5 17:32 memo2
-rw-r--r--    1 yury     yury       17 Feb  5 17:32 test1
```

Для архивирования файлов следует воспользоваться командой

```
$ ls | cpio -oc > arch
```

Извлечь все файлы из созданного архива можно с помощью команды

```
$ cat arch | cpio -icd
```

Если необходимо восстановить из архива только некоторые файлы, то следует их перечислить в команде `cpio`:

```
$ cat arch | cpio -icd "memo1" "memo2"
```

Здесь файлы memo1 и memo2 извлекаются из ранее созданного архива arch. Опция `-c` используется, если архив был создан с переносимым заголовком. Если путевые имена восстанавливаемых файлов содержат одинаковые символы, можно воспользоваться шаблоном:

```
$ cat arch | cpio -icd "m*"
```

В следующем примере выполняется копирование файлов в другой каталог. Для этого применяется команда `cpio -p`, которая принимает из стандартного входного потока список файлов и копирует или создает на них ссылки (опция `-l`) в другом каталоге (NEWDIR в данном случае). Опция `-d` требует создания каталогов при необходимости, а опция `-m` запрещает модификацию времени изменения файла.

Для генерации списка полных путевых имен файлов для `cpio` в команде `find` нужно задать опцию `-depth`. Это позволяет создавать файлы в каталогах, доступных только для чтения. Замечу, что целевой каталог NEWDIR должен существовать к моменту выполнения команды. Вот полная командная строка для выполнения операции копирования:

```
$ find . -depth -print | cpio -pdlmv NEWDIR
```

В следующем примере выполняется копирование файлов, находящихся в текущем каталоге и удовлетворяющих шаблону `t*`, в архивный файл TARCH:

```
$ find . -name 't*' -print|cpio -o  > TARCH
1 block
```

Восстановить файлы из архива TARCH можно при помощи командной строки

```
$ cpio -iuvB < TARCH
```

Для извлечения отдельных файлов из архива, созданного командой `cpio`, в командной строке следует указать имена файлов:

```
$ cpio -iuvB < TARCH "test2" "test3"
```

Здесь из архива TARCH извлекаются только файлы test2 и test3.

Используя опцию `-t`, можно просмотреть содержимое архива:

```
$ cpio -it < TARCH
```

Команда `cpio` по завершению формирует статус выхода. При этом значению 0 соответствует успешное завершение, если же значение статуса больше 0, это значит, что произошла ошибка.

Еще одной, очень популярной утилитой, позволяющей архивировать и восстанавливать файлы, является `tar`. Команда `tar` позволяет сохранять файлы на архивном носителе (например, дискете или ленте) и/или восстанавливать

их с данного носителя. Операции, выполняемые командой, определяются строкой символов, содержащей одну опцию (c, r, t, u или x) и, возможно, один или несколько модификаторов (v, w, f, b, L, k, F, X, h, i, e, n, A, l, m, o, p и num).

Остальные аргументы команды — имена файлов (или каталогов), указывающие, какие файлы необходимо заархивировать или извлечь из архива. Во всех случаях указание имени каталога означает ссылку на все файлы и (рекурсивно) подкаталоги этого каталога.

Команда принимает множество опций, наиболее часто используемыми из которых являются:

- -c — позволяет создавать новый архив, при этом запись начинается с начала архива;
- -r — позволяет добавлять указанные файлы в конец существующего архива;
- -t — имена и другая информация об указанных файлах выдается для каждого их вхождения в архив (задан модификатор v). При отсутствии модификатора v выдаются только имена файлов в формате, аналогичном формату команды ls -l, если же файлы не заданы, то выдается информация обо всех файлах в архиве;
- -u — позволяет добавить указанные файлы в архив, если их еще нет или если они изменились с момента последней записи в данный архив;
- -x — позволяет извлекать указанные файлы из архива, при этом если приведенное имя соответствует имени каталога, содержимое которого было записано в архив, то извлекается (рекурсивно) этот каталог. Желательно использовать относительный путь к файлу или каталогу, иначе tar не найдет его. Восстанавливаются также атрибуты файла (владелец, дата изменения и права доступа к файлу, если это возможно). Если файлы не указаны, извлекается все содержимое архива.

В процессе работы команда может выдавать сообщения об ошибках чтения/записи или о нехватке свободной памяти для размещения таблиц связей. Кроме того, существует ограничение на длину путевого имени файла, которая не должна превышать 100 символов.

Недостатком команды tar является и низкая скорость работы в режиме u, а опция b не может быть использована при работе с архивом, который должен обновляться. Если архив находится в дисковом файле, то опцию b нельзя применять ни в коем случае, потому что обновление архива, расположенного на диске, может разрушить его. Кроме того, команда tar не копирует пустые каталоги и специальные файлы.

Что же касается совместимости форматов архивов, созданных различными программами, то, например, утилита `cpio` распознает архивы, созданные при помощи `tar`, поэтому ее можно применять для чтения таких архивов.

Рассмотрим несколько примеров использования команды `tar`.

Предположим, что в текущем каталоге находятся такие файлы:

```
-rw-r--r--    1 root     root           10 Feb  8 01:09 test1
-rw-r--r--    1 root     root           19 Feb  8 01:09 test2
-rw-r--r--    1 root     root            0 Feb  8 01:10 test3
```

Для создания архива archive.tar, содержащего файлы test1, test2 и test3, нужно выполнить команду

```
# tar -cf archive.tar test1 test2 test3
```

Просмотреть содержимое созданного архива можно при помощи команды

```
# tar -tvf archive.tar
-rw-r--r-- root/root        10 2006-02-08 01:09:50 test1
-rw-r--r-- root/root        19 2006-02-08 01:09:58 test2
-rw-r--r-- root/root         0 2006-02-08 01:10:08 test3
```

Чтобы извлечь файлы из архива, нужно выполнить команду

```
# tar -xf archive.tar
```

Пусть необходимо скопировать содержимое текущего каталога в каталог ARCHTAR, который еще не создан. Особенностью команды `tar` является то, что, заданная с модификатором `f` и опцией "-", она может прочитать стандартный ввод и использоваться в конвейере команд. Следующая последовательность команд позволяет скопировать содержимое рабочего каталога в целевой каталог ARCHTAR:

```
$ tar cf - .|(mkdir ARCHTAR; cd ARCHTAR; tar xf -)
```

Для копирования нескольких каталогов в целевой каталог можно создать несложный командный файл:

```
for x in $*
 do
  tar cf - $x|(cd `pwd`/DEST; tar xf -)
 done
```

Здесь содержимое каталогов, заданных в качестве аргументов командной строки, копируется в каталог DEST, который к моменту копирования уже должен существовать.

5.6. Устройства в UNIX

Поскольку все устройства UNIX доступны посредством файлов устройств, то для них действительны все операции, выполняемые над файлами. Тем не менее, существует и определенная специфика работы с устройствами, которую необходимо учитывать при различных манипуляциях с этими объектами.

Все устройства операционной системы образуют так называемую подсистему ввода/вывода, связывающую работающие процессы с периферийными устройствами, такими как накопители на магнитных дисках и лентах, терминалы, принтеры и сети, с одной стороны и с модулями ядра, которые управляют устройствами и именуются драйверами устройств, с другой. Драйверы устройств управляют каким-либо одним типом устройств — в системе может быть один дисковый драйвер для управления всеми дисководами, один терминальный драйвер для управления всеми терминалами и один ленточный драйвер для управления всеми ленточными накопителями.

Если в системе имеются однотипные устройства от разных изготовителей, например, две марки ленточных накопителей, то они воспринимаются системой как устройства двух различных типов, что требует наличия двух отдельных драйверов из-за различных управляющих команд для этих устройств.

Кроме того, система поддерживает так называемые виртуальные устройства, с которыми не связано ни одно конкретное физическое устройство. Примером такого устройства может выступать физическая память. В этом есть определенный смысл, поскольку позволяет процессу обращаться к памяти как к устройству. Команда `ps`, например, обращается к информационным структурам ядра в физической памяти, чтобы сообщить статистику процессов.

Операционная система UNIX поддерживает два типа устройств — устройства ввода/вывода блоками и устройства неструктурированного или посимвольного ввода/вывода. Устройства ввода/вывода блоками или блочные устройства ввода/вывода, такие как диски и ленты, для остальной части системы выглядят как запоминающие устройства с произвольной выборкой. Устройства посимвольного ввода/вывода манипулируют с произвольным потоком данных в виде последовательности байтов, и к ним относятся все другие устройства, в том числе терминалы и сетевое оборудование.

Блочные устройства ввода/вывода могут иметь интерфейс и с устройствами посимвольного ввода/вывода. Как уже было упомянуто, пользовательские программы взаимодействуют с устройствами посредством специальных файлов устройств.

Специальный файл устройства имеет индекс и занимает определенное место в иерархии каталогов файловой системы. Файл устройства отличается от

других файлов типом, хранящимся в его индексе, в зависимости от устройства (блочное или символьное), которое он представляет. В тех случаях, когда устройство имеет как блочный, так и символьный интерфейс, для него определены два файла: специальный файл блочного устройства ввода/вывода и специальный файл устройства посимвольного ввода/вывода. Файлы блочных устройств обычно ссылаются на такие устройства, как, например, жесткие диски. Данные с этих устройств могут быть получены с помощью номера блока. Кроме того, такие устройства могут иметь и кэш-блоки.

Файлы устройств, так же как и обычные файлы, управляются системными вызовами `open()`, `close()`, `read()` и `write()`.

Файлы устройств в UNIX обозначаются специальным образом. Например, файл /dev/console соответствует терминалу консоли. Вывод, направленный в специальный файл /dev/console, будет появляться на экране терминала, а при попытке прочитать данные из файла /dev/console фактически будет прочитан символ с клавиатуры. Некоторые файлы устройств имеют особое назначение и ссылаются на сервисы операционной системы, например, /dev/null, /dev/random.

Для создания файлов устройств существует команда `mknod`, в которой указывается тип файла (блочный или символьный), а также старший и младший номера устройства. Замечу, что создавать файл устройства может только суперпользователь root. Команда `mknod` имеет такой синтаксис:

`mknod [`*опции*`] `*имя*` {bc} `*старший_номер младший_номер*

`mknod [`*опции*`] `*имя*` p`

Имя устройства должно быть задано в форме /dev/dev_name. Тип устройства указывается символом `c` (для символьных устройств с побайтовым доступом, как, например, последовательные порты) или символом `b` (для блочных устройств, с которыми возможен обмен данными блоками, например, дисковые накопители).

Следующий параметр представляет собой старший номер уникального идентификатора, характеризующего группу устройств (например, 4 — идентификатор виртуальных терминалов).

После старшего номера указывают младший номер устройства, представляющий собой идентификатор конкретного устройства в данной группе (например, младший номер 0 в группе старшего номера 4 — идентификатор первой системной виртуальной консоли, а 63 — идентификатор последней теоретически возможной из них).

Старший номер устройства показывает его тип, которому соответствует точка входа в таблице ключей устройств, младший номер устройства — это по-

рядковый номер единицы устройства данного типа. С одним старшим номером устройства может быть связано множество периферийных устройств, а младший номер устройства позволяет отличить их одно от другого. Не нужно создавать специальные файлы устройств при каждой загрузке системы; их только нужно корректировать, если изменилась конфигурация системы, например, если к установленной конфигурации были добавлены устройства.

Кроме указанных, команда `mknod` может принимать и дополнительные опции GNU:

```
[-m права] [--help] [--version] [--]
```

В следующем примере создается устройство /dev/tty10:

```
# mknod /dev/tty10 c 3 11
```

Здесь параметр `c` указывает на тип устройства (в данном случае символьное), `3` — старший номер устройства, а `11` — младший номер устройства.

Фактически команда `mknod` создает именованный канал — специальный файл символьного или блочного устройства с указанным именем. Созданный специальный файл не занимает места на диске и используется только для взаимодействия с операционной системой, но не для хранения данных.

GNU-версия `mknod` позволяет считать символ `u` синонимом типа `c`. Рассмотрим дополнительную опцию GNU, обозначаемую как `m` или `--mode`. Эта опция определяет права доступа к создаваемым файлам. Опция может быть указана в одной из двух форм:

```
-m права
--mode=права
```

Значение прав доступа к создаваемым файлам становится равным по величине значению аргумента *права*; оно может иметь как символьную форму, описанную в man-странице команды `chmod`, так и восьмеричную.

Приведу пример использования команды `mknod` с указанием атрибутов доступа:

```
$ mknod --mode=644 /dev/tty62 c 4 63
```

Эта команда создаст файл устройства для предпоследней теоретически возможной виртуальной консоли.

5.7. Программный интерфейс пользователя

Операционная система UNIX, помимо стандартных средств командной оболочки для работы с объектами файловой системы, включает целый ряд функций ядра для работы с файлами, доступных через системные вызовы. Мы уже

рассматривали использование системных вызовов при разработке утилит управления учетными записями, сейчас же проанализируем несколько примеров, разработанных на языке C++, позволяющих выполнять различные манипуляции с файлами.

Функции прикладного интерфейса программирования (API), представляющие собой системные вызовы операционной системы UNIX, дают возможность пользователям, знакомым с языком C или C++, манипулировать с файлами в своих программах. Кроме того (что очень важно) знание механизмов работы таких функций способствует более глубокому пониманию работы операционной системы в целом и правильному использованию ее возможностей. Учитывайте и то, что все команды операционной системы (`chmod`, `chown`, `cp`, `mv` и т. д.), выполняющие манипуляции файлами и доступные пользователю, созданы на основе системных вызовов, выполняющих обработку файлов на низком уровне.

В этом разделе представлены простые примеры манипулирования объектами файловой системы с помощью функций API, демонстрирующие основные принципы применения таких функций. Как правило, в этих демонстрационных программах не проводится обработка всех возможных ошибок, возникающих при работе с файлами, что сделано с целью упрощения исходных текстов. Пользователи, знакомые с программированием на C++, при желании могут усовершенствовать примеры программ и превратить их в полнофункциональные приложения.

Пример 1. Необходимо установить следующие атрибуты доступа к файлу test:

```
-rwx------    1 root     root            4 Jan 30 10:55 test
```

Исходный текст программы, выполняющей эту задачу, представлен далее:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(void)
{
 chmod ("test", S_IRUSR | S_IWUSR | S_IXUSR);
 return 0;
}
```

Здесь используется системный вызов `chmod()`, принимающий в качестве первого параметра путевое имя файла, а вторым параметром служит флаг доступа,

представляющий собой комбинацию макросов, определенных в файле заголовка sys/stat.h. Функция `chmod` имеет синтаксис:

```
int chmod(const char* path, mode_t flag)
```

Здесь *`path`* — путевое имя файла, а *`flag`* — права доступа. На базе этой функции реализована команда `chmod` операционной системы UNIX.

Пример 2. Необходимо программно установить флаг выполнения для остальных пользователей файла test. Предположим, что файл имеет следующие атрибуты:

```
-r--r--r--    1 root     root            4 Jan 30 10:55 test
```

Исходный текст программы (назовем ее change_perm), выполняющей установку соответствующего бита кода доступа, показан далее:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(void)
{
 struct stat statv;
 stat("test", &statv);
 int flag = statv.st_mode | S_IXOTH;
 chmod ("test", flag);
 return 0;
}
```

После выполнения программы атрибуты файла test будут такими:

```
# ./change_perm
# ls -l test
-r--r--r-x    1 root     root            4 Jan 30 10:55 test
```

В этой программе используется уже знакомая нам функция `chmod()`, и, кроме того, для получения атрибутов запрашиваемого файла должен быть выполнен системный вызов `stat()`. Функция `stat` имеет синтаксис:

```
int stat(const char* path, struct stat* statv)
```

Здесь *`path`* — путевое имя файла, а второй аргумент — это адрес переменной типа `struct stat`. Тип данных `struct stat` определен в заголовке `sys/stat.h`, а объявление самой структуры выглядит так:

```
struct stat
```

```
 {
   dev_ts   t_dev;       /* идентификатор файловой системы */
   ino_t    st_ino;      /* номер индексного дескриптора файла */
   mode_t   st_mode;     /* тип файла и флаги доступа */
   nlink_t  st_nlink;    /* значение счетчика жестких ссылок*/
   uid_t    st_uid;      /* идентификатор владельца файла*/
   gid_t    st_gid;      /* идентификатор группы*/
   dev_t    st_rdev;     /* старший и младший номера устройства*/
   off_t    st_size;     /* размер файла в байтах */
   time_t   st_atime;    /* время последнего доступа*/
   time_t   st_mtime;    /* время последней модификации*/
   time_t   st_ctime;    /* время последнего изменения статуса*/
};
```

Информация о файле test помещается в переменную `statv` после выполнения оператора

```
stat("test", &statv);
```

Нас будет интересовать поле `st_mode` структуры `statv`, поскольку оно содержит атрибуты доступа к файлу.

К существующим правам доступа следует добавить право на выполнение для остальных пользователей, что достигается путем логического сложения существующих прав доступа и макроса `S_IXOTH` в операторе:

```
int flag = statv.st_mode | S_IXOTH;
```

Наконец, оператор

```
chmod ("test", flag);
```

устанавливает требуемый код доступа.

Пример 3. Необходимо изменить права доступа для файла test на права записи для всех пользователей. Пусть файл test имеет атрибуты:

```
# ls -l test
-r--r--r--    1 root     root            4 Jan 30 10:55 test
```

Тогда для изменения прав доступа необходимо выполнить программу (назовем ее change_perm_1), исходный текст которой показан далее:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <unistd.h>

int main(void)
{
 struct stat statv;
 stat("test", &statv);
 int flag = S_IRUSR | S_IRGRP | S_IROTH;
 flag = (statv.st_mode & ~flag) | S_IWUSR | S_IWGRP | S_IWOTH;
 chmod ("test", flag);
 return 0;
}
```

Смысл операторов, встречающихся в этой программе, нам знаком из предыдущего примера. Обратите внимание на то, как формируется код доступа:

```
int flag = S_IRUSR | S_IRGRP | S_IROTH;
flag = (statv.st_mode & ~flag) | S_IWUSR | S_IWGRP | S_IWOTH;
```

После выполнения программы атрибуты файла test изменятся и будут такими:

```
# ./change_perm_1
# ls -l test
--w--w--w-    1 root     root            4 Jan 30 10:55 test
```

Пример 4. Необходимо сменить владельца нескольких файлов. Предположим, что имеются такие файлы:

```
# ls -l test*
--w--w--w-    1 yury     root            4 Jan 30 10:55 test
-rw-r--r--    1 yury     root            2 Feb  6 18:28 test1
-rw-r--r--    1 yury     root            3 Feb  6 18:28 test2
-rw-r--r--    1 yury     root            4 Feb  6 18:28 test3
```

Требуется сменить владельца файлов test1 и test2 на root.

Поможет это сделать программа (назовем ее chown_files), исходный текст которой показан далее:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <pwd.h>

int main(int argc, char* argv[])
{
 struct passwd *psw;
 struct stat statv;
 uid_t uid;

 if (argc < 3)
  {
   printf("Usage: %s new_owner file1 file2 ...\n");
   exit(1);
  }
 psw = getpwnam(argv[1]);
 uid = psw->pw_uid;
 if (uid == (uid_t)-1)
 {
  printf("Invalid user name\n");
  exit(2);
 }
 for (int i1 = 2; i1 < argc; i1++)
 {
  if (stat(argv[i1], &statv) == 0)
   {
    if (chown(argv[i1], uid, statv.st_gid))
     {
      printf("Cannot change owner for %s\n", argv[i1]);
     }
   }
 }
 return 0;
}
```

Программа принимает в качестве первого параметра регистрационное имя нового владельца, после которого может следовать переменное число параметров, представляющих собой путевые имена файлов.

В этой программе основную работу выполняет системный вызов `chown()`, позволяющий изменить идентификатор владельца и идентификатор группы данного файла и имеющий синтаксис:

```
int chown(const char* path, uid_t uid, gid_t gid)
```

Здесь *path* — путевое имя файла, *uid* — новый идентификатор владельца и *gid* — новый идентификатор группы. Если значение *uid* или *gid* равно –1, то соответствующий идентификатор файла не изменился.

Кроме того, нужны идентификаторы пользователя и группы, которые можно получить, вызвав функцию `getpwnam()`. Функция позволяет по регистрационному имени пользователя получить информацию о пользователе, записанную в переменную `struct passwd`. Вот синтаксис функции `getpwnam()`:

```
const struct passwd* getpwnam(const char* user_name)
```

Здесь *user_name* — регистрационное имя пользователя, а стуктура `struct passwd` определена в файле заголовка `pwd.h` следующим образом:

```
struct passwd
 {
   char*  pw_name;    /* регистрационное имя пользователя */
   char*  pw_passwd;  /* зашифрованный пароль */
   int    pw_uid;     /* идентификатор пользователя */
   int    pw_gid;     /* идентификатор группы */
   char*  pw_age;     /* минимальный срок действия пароля */
   char*  pw_comment; /* общая информация о пользователе */
   char*  pw_dir;     /* начальный (домашний) каталог пользователя */
   char*  pw_shell; /* регистрационный интерпретатор shell пользователя*/
};
```

Операторы

```
psw = getpwnam(argv[1]);
uid = psw->pw_uid;
```

сохраняют идентификатор пользователя в переменной `uid`, после чего в цикле `for` обрабатывается каждый файл. Вначале извлекаются его атрибуты при помощи функции `stat` (в операторе `if`):

```
stat(argv[i1], &statv)
```

затем функция `chown` (в операторе `if`) изменяет владельца файла:

```
chown(argv[i1], uid, statv.st_gid)
```

Обратите внимание на то, что владелец группы для данного файла не меняется.

Результат выполнения программы выглядит так:

```
# ./chown_files root test1 test2
# ls -l test*
```

```
--w--w--w-    1 yury     root            4 Jan 30 10:55 test
-rw-r--r--    1 root     root            2 Feb  6 18:28 test1
-rw-r--r--    1 root     root            3 Feb  6 18:28 test2
-rw-r--r--    1 yury     root            4 Feb  6 18:28 test3
```

Пример 5. Требуется вывести на консоль путевые имена файлов текущего каталога и их размер в байтах, а также суммарный размер файлов в байтах и количество файлов. Исходный текст программы представлен далее:

```
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>

int main(void)
{
 DIR* dir;
 struct dirent *fs;
 struct stat statv;
 int totalSize = 0;
 int countFiles = 0;

 //
 dir = opendir("./");
 while (fs = readdir(dir))
  {
   if ((strcmp(fs->d_name, ".") && strcmp(fs->d_name, "..")))
     {
      stat(fs->d_name, &statv);
      printf("%s\t\tSize:\t %d bytes\n", fs->d_name, statv.st_size);
      totalSize += statv.st_size;
      countFiles++;
     }
  }
 printf("Total number of files and directories: %d\n", countFiles);
 printf("Total size in bytes: %d\n", totalSize);
 closedir(dir);
 return 0;
}
```

Для работы с каталогами используются системные вызовы `opendir()` и `readdir()`. Функция `opendir()` является аналогом системного вызова `open()` для файлов. В качестве параметра она принимает путевое имя файла каталога и открывает этот файл только для чтения. Эта функция возвращает указатель на структуру `DIR*`, напоминающую структуру `FILE*`, возвращаемую функцией `fopen()`. Структура `DIR*` определена в заголовке `dirent.h` или в `sys/dir.h`.

Синтаксис функции `opendir()` таков:

```
DIR* opendir(const char* path)
```

Здесь *`path`* — путевое имя файла.

Еще одна функция, `readdir()`, читает следующую запись из файла каталога. В качестве аргумента функция принимает значение `DIR*`, возвращенное системным вызовом `opendir()`. Функция `readdir()` возвращает адрес записи типа `struct direct` или `struct dirent`, в которой хранится имя файла. При этом при первом вызове функция возвращает указатель на первую запись, при втором вызове — на вторую запись и т. д. Просмотрев все записи, `readdir()` возвращает нулевое значение.

Функция `readdir()` имеет синтаксис:

```
dirent* readdir(DIR* dir)
```

Таким образом, оператор

```
dir = opendir("./");
```

открывает текущий каталог и возвращает указатель `dir`. Далее, в цикле `while` функция `readdir()` считывает по одной записи, размещая ее в переменной `fs`. При помощи функции `stat()` (мы рассмотрели ее ранее) извлекаются атрибуты файла, путевое имя которого возвращается в переменной `fs->d_name`.

При успешном вызове функции `stat()` переменная `statv.st_size` содержит размер файла, который вместе с именем файла выводится на консоль функцией `printf()`. Переменная `countFiles` содержит количество файлов в каталоге, а переменная `totalSize` — суммарный размер файлов.

Оператор

```
if ((strcmp(fs->d_name, ".") && strcmp(fs->d_name, "..")))
```

позволяет исключить из обработки путевые имена текущего и родительского каталога.

Пример 6. Требуется создать копию файла. Следующая программа, исходный текст которой представлен далее, выполняет копирование файлов, причем

в качестве первого аргумента указывается исходный файл, а в качестве второго — файл назначения:

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
 int fdsrc, fddst;
 char buf[4096];
 int bytesRead;
 //
 if (argc != 3)
  {
   printf("Usage: %s src dst\n", argv[0]);
   exit(1);
  }
 if (access(argv[1], R_OK))
  {
   printf("File %s: access denied\n", argv[1]);
   exit(2);
  }
 fdsrc = open(argv[1], O_RDONLY, 0);
 fddst = open(argv[2], O_RDWR | O_CREAT, S_IRWXU | S_IRGRP | S_IROTH);
 if (fddst == -1)
  {
   printf("Error of creating %s\n", argv[2]);
   exit(3);
  }
 do {
      bytesRead = read(fdsrc, buf, sizeof(buf));
      if (bytesRead > 0)
       write(fddst, buf, bytesRead);
    } while (bytesRead > 0);
 printf("Done.\n");
 close(fdsrc);
```

```
 close(fddst);
 return 0;
}
```

В этой программе присутствуют системные вызовы, которые чаще других используются в файловых операциях: `open()`, `read()`, `write()`, `close()`. Функция `open()` имеет синтаксис:

```
int open(const char* path, int access_mode, mode_t permission)
```

Здесь *path* — путевое имя файла; *access_mode* — целочисленное значение, показывающее, какой тип доступа разрешен вызывающему процессу (значение может быть одним из макросов, определенных в заголовке `fcntl.h`); *permission* — целочисленное значение, которое необходимо только в том случае, если в аргументе *access_mode* установлен флаг `O_CREAT` (определен в виде макросов в заголовке `sys/stat.h`).

При успешном завершении системный вызов `open()` возвращает целочисленный дескриптор файла, по которому к файлу могут обращаться из других системных вызовов. В случае неудачи функция `open()` возвращает –1.

Системный вызов `read()` извлекает блок данных определенного размера из файла с указанным дескриптором. Функция имеет синтаксис:

```
ssize_t read(int fd, void* buf, size_t size)
```

Здесь *fd* — дескриптор открытого файла, *buf* — адрес буфера, в который будут помещены данные, а *size* — количество байтов, подлежащих чтению. Тип данных `ssize_t` определен в заголовке `sys/types.h` и эквивалентен типу `unsigned int`. При успешном завершении функция `read()` возвращает количество фактически прочитанных байтов.

Функция `write()` записывает блок данных определенного размера в открытый файл и имеет синтаксис:

```
ssize_t write(int fd, const void* buf, size_t size)
```

Системный вызов `write()` выполняет действие, обратное `read()`, при этом данные, размер которых определяется параметром *size*, берутся из области памяти *buf* и записываются в файл, дескриптор которого равен *fd*. При успешном завершении функция возвращает фактическое количество записанных байтов.

Наконец, системный вызов `close()` отсоединяет файл от вызвавшего его процесса:

```
int close(int fd)
```

Здесь *fd* — дескриптор файла. В случае успешного завершения функция возвращает 0, в противном случае –1.

Вернемся к исходному тексту программы. Перед операцией копирования проверяется, имеет ли процесс доступ по чтению к исходному файлу. Это выполняет оператор

```
if (access(argv[1], R_OK))
```

в котором указан системный вызов `access()`. В качестве путевого имени исходного файла используется первый аргумент программы. Если файл доступен, то он открывается при помощи системного вызова `open()`, реализованного в операторе

```
fdsrc = open(argv[1], O_RDONLY, 0);
```

В случае успешного выполнения возвращается дескриптор `fdsrc` открытого файла. Файл назначения (куда будет выполняться копирование) создается при помощи системного вызова `open()` оператором

```
fddst = open(argv[2], O_RDWR | O_CREAT, S_IRWXU | S_IRGRP | S_IROTH);
```

Здесь `argv[2]` — путевое имя файла, который должен быть создан и открыт. Если вызов завершается успешно, то возвращается дескриптор `fddst` открытого файла назначения.

Далее, в цикле `do` выполняется собственно копирование содержимого файла-источника в файл назначения. Наконец, после завершения операции копирования следует закрыть дескрипторы файлов при помощи системных вызовов `close()`:

```
close(fdsrc);
close(fddst);
```

Пример 7. Требуется создать жесткую ссылку (hard link) на существующий файл. Для решения этой задачи воспользуемся системным вызовом `link()`. Программа, исходный текст которой представлен далее, в качестве первого аргумента принимает путевое имя существующего файла, а в качестве второго — создаваемую ссылку:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
 if(argc != 3 || !strcmp(argv[1], argv[2]))
  {
   printf("Usage: %s old_link new_link\n", argv[0]);
```

```
   exit(1);
  }
 if (access(argv[1], F_OK))
   {
    printf("%s: access denied.\n", argv[1]);
    exit(2);
   }
 if (link(argv[1], argv[2]) == -1)
   {
     printf("Cannot create link to %s\n", argv[1]);
     exit(3);
   }
 return 0;
}
```

Функция `link()` имеет такой синтаксис:

```
int link(const char* old_link, const char* new_link)
```

Здесь *`old_link`* — путевое имя существующего файла, а *`new_link`* — новое путевое имя, которое должно быть присвоено этому же файлу. При успешном выполнении вызова счетчик жестких ссылок файла увеличивается на единицу.

Что же касается исходного текста программы, то здесь оператор

```
if(argc != 3 || !strcmp(argv[1], argv[2]))
```

препятствует попытке дублирования жесткой ссылки, а оператор

```
if (access(argv[1], F_OK))
```

в котором используется системный вызов `access()`, проверяет существование файла с указанным путевым именем.

Пример 8. Воспользуемся предыдущим примером и создадим программу для перемещения файлов (наподобие `mv`). В этой программе, как и в предыдущей, используется системный вызов `link()`, и, кроме того, функция `unlink()` будет удалять исходное имя файла. Вот исходный текст программы:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char* argv[])
```

```
{
 if(argc != 3 || !strcmp(argv[1], argv[2]))
  {
   printf("Usage: %s old_path new_path\n", argv[0]);
   exit(1);
  }
 if (access(argv[1], F_OK))
   {
    printf("%s: access denied.\n", argv[1]);
    exit(2);
   }
 if (link(argv[1], argv[2]) == -1)
   {
     printf("Cannot create file %s\n", argv[1]);
     exit(3);
   }
 if (unlink(argv[1]) == -1)
  {
   printf("Cannot delete file %s\n!", argv[1]);
   exit(4);
  }
 return 0;
}
```

В этом примере, кроме уже знакомых нам функций `access()` и `link()`, используется системный вызов `unlink()`. Функция `unlink()` удаляет ссылку на существующий файл, уменьшая значение счетчика ссылок и удаляя соответствующую запись из каталога. При успешном выполнении функции к данному файлу уже нельзя будет обратиться с помощью этой ссылки. Файл удаляется из файловой системы, когда счетчик его жестких ссылок становится равным нулю, и ни один процесс не использует дескриптор, обозначающий этот файл.

Функция имеет синтаксис:

```
int unlink(const char* link_to_file)
```

Здесь *link_to_file* — путевое имя существующего файла. Если вызов функции завершается успешно, то возвращается 0, в противном случае возвращается –1.

В остальной части исходный текст похож на тот, что приведен в *примере 7*, поэтому останавливаться на нем мы не будем.

Пример 9. Здесь демонстрируется работа с устройствами ввода/вывода, такими как терминальные линии операционной системы Linux. Каждому открытому окну консоли соответствует терминальное устройство. Например, если открыты два окна, то им соответствуют файлы устройств /dev/pts/0 и /dev/pts/1, в которые можно записывать и из которых можно читать данные. Эти устройства — байт-ориентированные, поэтому допускают посимвольный ввод/вывод. Поскольку в операционной системе UNIX для работы с файлами имеются системные вызовы `open()`, `read()`, `write()` и `close()`, то они могут с успехом использоваться в операциях ввода/вывода для устройств.

Следующая программа (она называется write_to_dev), исходный текст которой представлен далее, демонстрирует запись данных в терминальную линию, при этом имя файла устройства является единственным параметром программы:

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
 char *buf = "Test string for TERMINAL";
 if (argc != 2)
 {
  printf("Usage: %s terminal\n", argv[0]);
  exit(1);
 }

 int fd = open(argv[1], O_RDWR);
 if (fd == -1)
 {
  printf("Cannot open terminal device %s \n", argv[1]);
  exit(2);
 }
 write(fd, buf, strlen(buf));
 close(fd);
 return 0;
}
```

Исходный текст программы несложен, поэтому ограничусь коротким комментарием. При помощи оператора

```
int fd = open(argv[1], O_RDWR);
```

выполняется попытка открыть терминальное устройство для чтения/записи и присвоить ему дескриптор `fd`. Если операция завершена успешно, то выполняется запись строки `buf` в устройство при помощи системного вызова `write()`:

```
write(fd, buf, strlen(buf));
```

после чего следует закрыть дескриптор устройства, что и делается посредством системного вызова `close()` в операторе `close(fd)`.

Вот результат работы программы при выводе строки на терминал, соответствующей устройству /dev/pts/0:

```
# ./write_to_dev /dev/pts/0
Test string for TERMINAL#
```

Все рассмотренные примеры демонстрируют только небольшую часть возможностей прикладного интерфейса (API) программирования операционной системы UNIX для работы с файлами. При желании читатель может более детально изучить функции API, довольно хорошо описанные в многочисленной документации.

Глава 6

Обработка текста в UNIX

Любой пользователь, независимо от того, в какой операционной системе он работает, время от времени сталкивается с необходимостью работы с текстовыми данными. Операционные системы UNIX на сегодняшний день содержит в себе развитые средства работы с любыми документами, включая и текст. Среди наиболее известных пакетов программ для комплексной обработки данных можно выделить StarOffice, позволяющий манипулировать различными типами данных. Кроме того, любая графическая оболочка (GNOME, KDE и др.) включает в себя, как минимум, несколько программ для работы с данными, в том числе и текстовыми. Упомянутые инструменты обработки данных предоставляют пользователю удобный графический интерфейс, что делает работу с ними легкой и приятной.

Тем не менее, в UNIX-системах достаточно широко используются и специальные программы для обработки текстовой информации, называемые текстовыми редакторами. Некоторые программы для обработки текста, такие как текстовый редактор `vi` и его усовершенствованная модификация `vim`, являются классическими приложениями для UNIX. Эти текстовые редакторы были разработаны и развивались одновременно с развитием операционных систем и включены в дистрибутивный пакет всех без исключения версий UNIX. Они являются незаменимыми инструментами при отладке системы в режиме текстовой консоли, когда требуется вносить изменения в файлы конфигурации, а графическая система X Window недоступна.

Еще одним классическим текстовым редактором является `emacs`. Он может работать как в текстовом, так и в графическом режимах. Как и в `vi`, в `emacs` для редактирования используются комбинации клавиш, причем любая команда может быть выполнена как с помощью мыши, так и с помощью клавиатуры. Редактор `emacs` обладает широкими возможностями для настройки, но, к сожалению, в настоящее время значительно уступает по удобству работы и функциональности программам с графическим интерфейсом.

Современные операционные системы, особенно их свободно распространяемые реализации, такие как Linux и FreeBSD, в качестве графических оболочек используют, как правило, GNOME и KDE. Для данных оболочек разработано много текстовых редакторов с графическим интерфейсом, и среди них наиболее распространенными являются программы `gedit` и `Kate`.

Программа `gedit` — текстовый редактор для графической оболочки GNOME. Он позволяет обрабатывать различные форматы файлов (HTML, Perl, C, PHP и др.), обеспечивая возможности для предварительного просмотра, отмены внесенных изменений и редактирования в нескольких закладках. Кроме того, `gedit` позволяет подключать к редактируемым файлам другие программы, так называемые плагины (plugin или plug-in), что значительно улучшает его функциональность.

Программа `Kate` — текстовый редактор оболочки KDE. Как и `gedit`, он также подходит для работы с различными языками программирования. Этот редактор использует разделенные экраны (split screens). Кроме того, из программы `Kate` можно открыть терминальное окно. Ко всему сказанному следует добавить, что редактор `Kate` интегрирован с Web-браузером Konqueror.

Должен сказать, что перечисленными программами для работы с текстовыми документами далеко не исчерпывается список популярных текстовых редакторов. Тем не менее, мы детально рассмотрим именно эти программы (`vi`, `gedit` и `Kate`) по двум основным причинам. Во-первых, они широко применяются в большинстве операционных систем, а во-вторых, остальные приложения данного класса по своим функциональным возможностям и принципам настройки очень похожи на перечисленные программы. Знание принципов функционирования этих трех программ позволит легко освоить другие приложения для обработки текста, учитывая в особенности и то, что никаких кардинальных отличий среди подобных программ не существует.

Рассмотрим возможности текстовых редакторов `vi`, `gedit` и `Kate` более подробно и начнем с классического текстового редактора `vi`.

6.1. Редактор *vi*

Редактор `vi` является мощным средством для создания и редактирования файлов и работает в оконном режиме. Строки редактируемого текста отображаются на экране, а изменения в тексте выполняются посредством нескольких простых команд. С помощью команд текстового редактора можно передвигать курсор в любую точку на экране или в файле, создавать, изменять или удалять текст.

Для создания нового файла в редакторе нужно набрать команду

```
# vi имя_файла
```

Например, если ввести команду `vi newfile`, редактор очистит экран и отобразит окно, в котором можно вводить и редактировать текст (рис. 6.1).

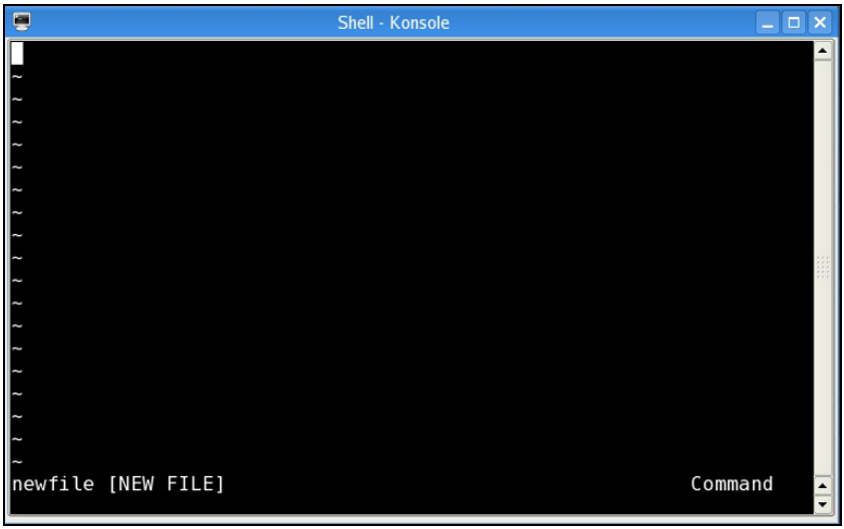


Рис. 6.1. Запуск редактора `vi`

6.1.1. Команды редактора *vi*

Редактор `vi` может работать в режиме ввода или командном режиме. В режиме ввода можно добавлять и редактировать текст. Что же касается командного режима, то он позволяет редактировать ранее введенный текст, включая удаление, перемещение и копирование, а также передвигаться по открытому для редактирования файлу и т. д.

При запуске редактора `vi` по умолчанию устанавливается командный режим, при котором `vi` ожидает ввода команд. Например, для ввода текста в открытый или вновь созданный файл необходимо выполнить такую последовательность шагов:

1. Нажать клавишу с латинской литерой <a> (это позволяет добавить текст к открытому файлу).
2. Ввести текст, при этом для перехода на новую строку нужно нажимать клавишу <Enter>.

По окончанию ввода следует нажать клавишу <Esc> — переход в командный режим, в котором текст можно редактировать. Для этих целей в редакторе `vi` предусмотрен целый ряд команд, которые можно выполнить, нажав определенные клавиши:

- <h> — выполняет перемещение курсора на одну позицию влево;
- <j> — выполняет перемещение курсора вниз на одну строку;
- <k> — выполняет перемещение курсора вверх на одну строку;
- <l> — выполняет перемещение курсора на одну позицию вправо.

Команды, инициируемые клавишами <j> и <k>, не влияют на позицию курсора по вертикали. Если количество символов в строке, куда переместился курсор, меньше, чем в предыдущей, то он устанавливается на последний символ этой строки.

Перемещение можно выполнять на несколько позиций. Если нажать, например, последовательность клавиш <5><l>, то курсор переместится на 5 позиций вправо. При этом, если невозможно его переместить на указанное число позиций (например, достигнут конец строки), то подается звуковой сигнал, и курсор устанавливается на последнюю возможную позицию.

Кроме клавиш <h> и <l>, переместить курсор влево/вправо можно соответственно клавишами <Backspace> и <Spacebar>, причем для перемещения на определенное число позиций следует ввести его с клавиатуры. Например, для перемещения курсора на 3 позиции вправо следует ввести <3><Spacebar>.

Для удаления символа нужно переместить курсор на соответствующую позицию и нажать клавишу <x>, при этом предварительно можно указать число символов, подлежащих удалению. Например, ввод команды <3><x> приведет к удалению 3-х символов подряд, начиная с позиции курсора. Следует отметить, что редактор `vi` корректирует строку, удаляя лишние пробелы после удаленных символов.

Добавить символы в редактируемый текст можно с помощью нажатия клавиши <i> или <a>. Для этого следует переместить курсор в нужную позицию при помощи одной из команд <h>, <j>, <k> или <l>, после чего нажать клавишу <i> и ввести текст. Введенный текст появляется на экране слева от позиции курсора. Точно так же для ввода текста можно использовать командную клавишу <a>, при этом введенный текст будет находиться справа от позиции курсора. При вводе текста вводимые символы будут продвигать весь текст вправо. Для окончания ввода следует нажать клавишу <Esc>.

Кроме клавиш <h>, <j>, <k>, <l>, <Backspace> и <Spacebar>, для перемещения курсора можно использовать и другие команды, которые называются

командами быстрого перемещения. Эти команды, как и предыдущие, выполняются при нажатии определенных клавиш, перечисленных далее:

- <$> — выполняет перемещение курсора на позицию, занимаемую последним символом в строке;
- <0> — выполняет перемещение курсора на позицию, занимаемую первым символом в строке;
- <^> — выполняет перемещение курсора на позицию, занимаемую первым ненулевым символом в строке.

Перемещение курсора в нужную позицию можно осуществлять, указав критерий поиска. Например, можно указать, что курсор следует установить перед каким-либо конкретным символом в строке. В редакторе `vi` предусмотрен целый ряд команд, позволяющих перемещать курсор, задав подобные условия. Вот перечень этих команд:

- <f> `x` — поиск символа `x` справа от текущей позиции курсора;
- <F> `x` — поиск указанного символа `x` слева от текущей позиции курсора;
- <t> `x` — перемещение курсора вправо на символ до указанного символа `x`;
- <T> `x` — перемещение курсора влево на символ до указанного символа `x`;
- <;> — повторение предыдущего поиска символа с запоминанием символа и осуществлением поиска следующего вхождения символа в текущей строке;
- <,> — повторение предыдущего поиска символа, но в обратном направлении. При этом символ запоминается, после чего осуществляется поиск следующего вхождения символа в текущей строке.

До сих пор мы рассматривали, в основном, перемещение курсора в строке. Для перемещения курсора по вертикали, т. е. по строкам, в редакторе `vi` предусмотрено несколько команд, перечисленных далее. Напомню, что все команды реализованы как нажатия клавиш, когда редактор `vi` работает в режиме ввода команд:

- <-> — выполняет перемещение курсора на одну строку вверх в позицию, где находится первый непустой символ. Для перемещения курсора на несколько строк нужно указать перед символом "-" их количество. Например, переместить курсор вверх на 7 строк можно, выполнив команду <7><->. Данный тип команды позволяет быстро перемещаться по буферу экрана, при этом если достигнуто начало буфера, т. е. первая строка, то курсор перемещаться не будет. Если часть строк находится вне видимой части экрана, то будет выполняться прокрутка текста;

- <+> — выполняет перемещение курсора на одну строку вниз в позицию, где находится первый непустой символ. Для перемещения курсора на несколько строк нужно указать перед символом "+" их количество. Например, переместить курсор вниз на 3 строки можно, выполнив команду <3><+>. Данный тип команды позволяет быстро перемещаться по буферу экрана, при этом если достигнут конец буфера, т. е. последняя строка, то курсор перемещаться не будет. Если часть строк находится вне видимой части экрана, то будет выполняться прокрутка текста.

Далее рассмотрим возможности редактора `vi` по обработке отдельных слов, представляющих собой строки символов, включая литеры, цифры и символы подчеркивания. Слова разделяются пустым пространством, которое состоит из символов пробела, табуляции и новой строки. Существует 6 команд позиционирования, реализованных посредством клавиш <w>, <b>, <e>, <W>, <B> и <E>. При этом команды <w>, <b>, <e> интерпретируют любой символ, отличный от литеры, цифры или подчеркивания как разделитель (символ начала или конца строки также является разделителем). Перечисленные команды выполняют следующие действия:

- <w> — перемещает курсор к началу следующего слова;
- *n*<w> — перемещает курсор к началу *n*-го слова, при этом, если обнаружен конец строки, то курсор переходит на следующую строку;
- <W> — перемещает курсор на следующее после пробела слово, игнорируя знаки пунктуации;
- <e> — перемещает курсор на позицию последнего символа в следующем слове;
- <E> — перемещает курсор на позицию последнего символа в слове, игнорируя все символы пунктуации, кроме пробела. В качестве разделителя слов должен использоваться пробел;
- <b> — перемещает курсор на позицию первого символа предыдущего слова;
- *n*<b> — перемещает курсор на позицию первого символа *n*-го предыдущего слова. В случае достижения начала строки выполняется переход в конец предыдущей строки.

Команда <B> выполняет те же действия, что и <b>, за исключением того, что слова разделяются символами пробела и новой строки, при этом все остальные символы пунктуации рассматриваются как литеры.

Редактор `vi` позволяет позиционировать курсор в начало определенного предложения. Для выполнения таких операций необходимо каким-то образом

распознавать конец предложения. Редактор `vi` распознает конец предложения, если оно заканчивается одним из символов: `!`, `.`, `?`. Если эти символы появляются в середине строки, то за ними должны следовать два пробела — только в этом случае `vi` сможет правильно их интерпретировать.

Перемещение курсора к началу предложения можно выполнить, нажав одну из клавиш: <(> или <)>. Вот смысл этих команд:

- <(> — перемещает курсор в начало текущего предложения;
- *n*<(> — перемещает курсор в начало *n*-го предложения назад от текущего предложения;
- <)> — перемещает курсор в начало следующего предложения;
- *n*<)> — перемещает курсор в начало *n*-го по счету предложения после текущего.

Что же касается работы с абзацами, то они распознаются редактором `vi` в том случае, если начинаются после пустой строки, поэтому для обеспечения такой возможности в конец каждого абзаца следует вставлять пустую строку. Имеется несколько команд для манипуляций с абзацами:

- <{> — перемещает курсор в начало текущего абзаца;
- *n*<{> — перемещает курсор в начало *n*-го после текущего абзаца;
- <}> — перемещает курсор в начало следующего абзаца;
- *n*<}> — перемещает курсор в начало *n*-го абзаца назад от текущей строки.

Еще 3 команды редактора `vi` позволяют позиционировать курсор в окне:

- <H> — курсор перемещается к первой строке экрана;
- <M> — курсор перемещается к средней строке экрана;
- <L> — курсор перемещается к последней строке экрана.

Большинство рассмотренных нами команд оперируют с текстом, находящимся в видимой части экрана. Для обработки остального текста следует вывести его на экран, установив курсор в соответствующую позицию. В практическом плане нужно выполнить одну из операций:

- выполнить прокрутку (скроллинг) содержимого файла вперед или назад;
- перейти к указанной строке файла;
- выполнить поиск по шаблону.

Прокрутку текста можно выполнить с помощью одной из четырех команд редактора `vi`:

- <Ctrl>+<f> — выполняет прокрутку экрана вперед на одно текстовое окно, расположенное ниже текущего окна, при этом редактор `vi` очищает

экран и выводит новое окно. Две последние строки текущего окна отображаются в верхней части нового окна. Если окно дополнено пустыми строками, то они помечаются символом ~;

- <Ctrl>+<d> — выполняет прокрутку экрана вперед на половину экрана, чтобы отобразить текст, расположенный ниже окна;
- <Ctrl>+<b> — осуществляет прокрутку экрана назад на полное окно, чтобы отобразить текст, который находится выше текущего экрана, при этом редактор `vi` очищает экран и отображает новое окно. В отличие от команды <Ctrl>+<f> эта команда не оставляет в новом окне строки из текущего окна;
- <Ctrl>+<u> — выполняет прокрутку половины экрана, чтобы отобразить текст, расположенный выше окна.

Следующая команда <G> устанавливает курсор на указанную строку в окне. Возможные модификации этой команды:

- <G> — курсор перемещается на последнюю строку в файле;
- *n*<G> — курсор перемещается на строку с номером *n*. Если строка отсутствует на текущем экране, то выполняется очистка экрана и отображается окно, содержащее эту строку.

Каждая строка имеет номер, соответствующий ее положению в буфере. Получить номер конкретной строки можно, переместив курсор на нее, после чего ввести команду <Ctrl>+<g>. В результате выполнения команды в нижней части экрана будет отображена следующая информация:

- имя файла;
- произошла ли модификация буфера после последней его записи в файл;
- номер строки, где находится курсор;
- общее количество строк в буфере.

Редактор `vi` включает в себя команды, с помощью которых можно выполнить поиск символов по шаблону. К ним относятся `/`, `?`, `n` и `N`. Они позволяют осуществлять поиск в буфере следующего вхождения символов, указанных в шаблоне, причем команды `n` и `N` позволяют повторить последнюю операцию поиска. Вот более подробное описание этих команд:

- </>*шаблон* — выполняет поиск вперед по буферу следующего появления символов, указанных в шаблоне, и помещает курсор на первый символ из шаблона. Например, командная строка

  ```
  /string
  ```

  пытается обнаружить в буфере первое вхождение слова `string` и помещает курсор под литерой `s`;

- <?>*шаблон* — выполняет поиск в обратном направлении в буфере первого появления символов, указанных в шаблоне, и помещает курсор на первый из них. Например, командная строка

  ```
  ?announce
  ```

  находит предыдущее вхождение в буфер слова `announce` и устанавливает курсор под литерой `a`.

Перечисленные команды не выполняют возврат в начало цикла в случае окончания строки во время поиска слов, хотя и выполняют циклический возврат в конце или начале буфера для продолжения поиска. Например, если курсор находится почти в конце буфера, а последовательность символов, соответствующих шаблону, — в начале буфера, то команда `/` обнаружит такое выражение.

Команды `n` и `N` позволяют повторить предыдущую команду поиска `/` или `?`, при этом `N` указывает на то, что поиск выполняется в обратном направлении.

Рассмотрим более подробно команды, позволяющие вводить и редактировать текстовую информацию (напомню, что все команды будут выполняться только при переходе редактора `vi` в командный режим посредством нажатия клавиши <Esc>):

- <a> — добавление текста после курсора;
- <A> — добавление текста в конец текущей строки;
- <i> — вставка текста до позиции курсора;
- <I> — вставка текста в начало текущей строки до первого ненулевого символа;
- <o> — создание текста в новой строке, которая будет располагаться ниже текущей строки. Этой командой можно воспользоваться в любом месте текущей строки;
- <O> — создание текста в новой строке, которая будет располагаться выше текущей строки. Этой командой можно воспользоваться в любом месте текущей строки.

При создании новой строки (команды <o> и <O>) редактор переводится в режим ввода текста, при этом курсор располагается в позиции, соответствующей началу новой строки.

Что же касается удаления текста, то во многих случаях можно использовать клавишу <Backspace>, удаляющую символ слева от позиции курсора. При этом курсор перемещается на одну позицию назад.

Кроме <Backspace>, удалить текст можно при помощи следующих комбинаций клавиш:

- <Ctrl>+<w> — отмена ввода текущего слова. При этом курсор возвращается к позиции первого символа последнего введенного слова. Слово не стирается с экрана до тех пор, пока не будет нажата клавиша <Esc> или не будут введены новые символы на месте этого слова;
- <Ctrl>+<v> — удаление специального значения (если оно имеется) следующих вводимых символов.

Для отмены последней введенной команды можно воспользоваться одной из двух команд:

- <u> — отменяет последнюю команду;
- <U> — восстанавливает текущую строку до состояния перед произведенными изменениями.

Например, при случайном удалении строк для отмены команды удаления можно ввести команду <u>, при этом удаленные строки будут восстановлены на экране. Что же касается команды <U>, то она отменяет все изменения текущей строки в том случае, если курсор находится в этой строке.

Повторный ввод команды <u> приведет к тому, что вторая команда отменит первую. Например, если удаленная строка была восстановлена командой <u>, то повторный ввод <u> вновь удаляет строку.

Удалить слово или часть слова можно с помощью команды <d>, для чего нужно переместить курсор к первому удаляемому символу и ввести команду <d><w>. В этом случае символ, на который указывает курсор, и все последующие будут стерты. Команда <d><w> удаляет одно слово или знак пунктуации, а также следующие за ним пробелы. Вы можете удалить несколько слов или знаков пунктуации за один раз, если укажете перед командой соответствующее число. Например, чтобы удалить 3 слова, нужно набрать команду <3><d><w>.

Для удаления целого абзаца можно воспользоваться командой <d><{> или <d><}>, а удалить целую строку можно с помощью команды <d><d>. Если требуется удалить несколько строк, то перед данной командой нужно ввести число строк, подлежащих удалению. Например, команда <5><d><d> удалит 5 строк.

Чтобы удалить весь текст в строке, расположенный после курсора, нужно поместить курсор на позицию первого удаляемого символа и ввести команду <D> или <d><$>. Замечу, что эти команды позволяют удалить текст только в текущей строке.

Редактирование и замена текста в `vi` выполняются посредством команд замены, подстановки и изменения текста. Вначале рассмотрим команды замены:

- <r>`x` — выполняет замену символа, на который указывает курсор, на `x`. Команда не переключает редактор в режим ввода текста, поэтому нет необходимости нажимать после нее клавишу <Esc>;
- *n*<r>`x` — выполняет замену *n* символов на `x` (нажатие клавиши <Esc> не требуется);
- <R> — выполняет замену только символов, введенных после нажатия клавиши <Esc>. При достижении конца строки ввод интерпретируется как новый текст.

Вот примеры замены текста. Если нужно заменить слово `string` на `String` в предложении

```
This is a first string in file.
```

то следует позиционировать курсор под литерой `s` в слове `string` и ввести команду <r><S>. После этого предложение будет выглядеть как

```
This is a first String in file.
```

Для замены текста "`This is a `" пробелами в предложении из предыдущего примера следует выполнить команду 10<r><Spacebar>. В результате останется предложение

```
          first String in file.
```

Команды подстановки текста заменяют символы, после чего ввод текста продолжается с той позиции, в которой была нажата клавиша <Esc>. К этой группе команд относятся:

- <s> — позволяет удалить символ, на который указывает курсор, и добавить текст. По окончанию ввода текста нужно нажать клавишу <Esc>;
- *n*<s> — позволяет удалить *n* символов и добавить текст. По окончанию ввода текста следует нажать клавишу <Esc>;
- <S> — позволяет заменить все символы в строке.

При вводе команды <s> последний символ в строке, подлежащий замене, заменяется символом `$`, причем он не убирается с экрана, пока вместо него не будет введен новый символ, или не будет нажата клавиша <Esc>.

Команды изменения <c> заменяют текст и затем продолжают добавлять текст с этого места до тех пор, пока не будет нажата клавиша <Esc>. Команда изменения позволяет заменить символ, слово и т. д. Вот синтаксис команд замены:

- *n*<c>`x` — позволяет заменить *n*-й объект текста типа `x` (предложение, абзац);

- <c>w — позволяет заменить слово или оставшиеся символы в слове на новый текст, при этом выводится символ $, указывающий на последний символ, который будет заменен;
- *n*<c>w — позволяет заменить *n* слов;
- c<c> — позволяет заменить все символы в строке;
- *n*<c>c — позволяет заменить все символы в текущей строке и в *n* верхних строках;
- <C> — позволяет заменить оставшиеся символы в строке от позиции курсора и до конца строки.

Команда <c> использует символ $, отмечая последний символ, который следует заменить. После выполнения команды изменения редактор vi находится в режиме ввода текста, поэтому можно продолжать вводить текст. Он будет находиться в буфере до нажатия клавиши <Esc>.

Рассмотрим команды редактора vi, которые позволяют перемещать и копировать фрагменты текста из одного места в другое. Есть несколько способов перемещения и копирования текста.

Например, можно переместить текст из одного места в буфере в другое, удалив строки, а затем поместить их в требуемое место. Здесь используется то обстоятельство, что удаленный текст сохраняется во временном буфере, поэтому достаточно переместить курсор на позицию, куда нужно поместить текст, и нажать клавишу <p>. При этом удаленные строки будут добавлены ниже текущей строки.

Кроме того, удаленную строку можно поместить в другую строку. Для этого нужно установить курсор между двумя словами, после чего нажать клавишу <p>. В результате удаленная строка будет помещена в позицию сразу после курсора.

Здесь следует учитывать, что временный буфер сохраняет результат только последней команды, поэтому команда <p> должна применяться сразу же после команды удаления. Следует отметить, что команда <p> используется и для копирования текста, помещенного во временный буфер командой <y>.

Для быстрого перемещения символов можно использовать комбинацию <x><p>. Команда <x> удаляет символ, а <p> помещает ее после следующего символа.

Следующая группа команд, которую мы рассмотрим, выполняет несколько весьма полезных при редактировании текста операций. К этим командам относятся:

- <.> — повторяет последнюю выполненную команду;

- <J> — объединяет две строки;
- <Ctrl>+<l> — очищает экран, вновь отображая его содержимое;
- <~> — заменяет нижний регистр литер на верхний и наоборот.

Команда <.> повторяет последнюю введенную команду. Наиболее часто эта команда используется для повторения поиска.

Команда <J> объединяет строки. Для этого нужно поместить курсор на текущей строке и нажать <Shift>+<j>. В результате текущая строка объединяется со следующей после нее строкой, при этом редактор `vi` автоматически ставит пробел между последним словом в первой строке и первым словом второй строки.

Для быстрой замены литеры нижнего регистра на символ верхнего регистра или наоборот нужно поместить курсор под этой литерой и нажать клавишу <~>. Таким образом можно заменить подряд несколько литер, если нажать несколько раз <~>, либо предварительно указав перед этой командой количество требуемых замен.

6.1.2. Сохранение текста и выход из редактора *vi*

По окончании работы с текстовыми данными необходимо записать содержимое буфера в файл, после чего возвратить управление командной оболочке shell. Для этого следует нажать комбинацию клавиш <Shift>+<z><z>. При этом редактор сохраняет информацию из буфера в файле, имя которого было указано в начале сеанса редактирования.

Для записи буфера в файл и выхода из `vi` можно воспользоваться также командами `:w` и `:q` редактора. Обратите внимание на то, что команды редактора начинаются с двоеточия : и появляются в последней строке экрана. Команда `:w` записывает буфер в файл, а `:q` завершает работу редактора, возвращая управление интерпретатору shell. Обе команды можно ввести одновременно в виде одной команды `:wq`.

Если в буфере редактируемого файла не было сделано никаких изменений, то завершить работу редактора можно командой `:q`, в противном случае `vi` выдаст предупреждающее сообщение о необходимости сохранения буфера или возможности завершения работы командой `:q!`. Команда `:q!` завершает работу с редактором `vi` без записи содержимого буфера в файл, при этом все изменения в буфере игнорируются.

При необходимости замены содержимого существующего файла на содержимое буфера следует воспользоваться командой :w!, например:

```
:w! имя_файла
```

Здесь *имя_файла* — имя перезаписываемого файла.

Иногда возникают неприятные ситуации, когда требуется восстановить редактируемый файл при внезапном прерывании работы редактора vi. Если это произошло, то временный буфер с данными не записывается в файл, но, несмотря на это, vi сохраняет копию буфера во временном файле. Это позволяет восстановить файл с помощью опции -r, заданной при запуске редактора vi, например:

```
vi -r имя_файла
```

При этом большинство выполненных до сбоя изменений будут загружены в буфер редактора vi, после чего можно продолжить редактирование файла или же сохранить буфер в файле.

Редактор предусматривает возможность обработки нескольких файлов в одном сеансе. Например, для редактирования файлов с именами file1 и file2 нужно ввести команду:

```
vi file1 file2
```

При этом vi отображает на экране информацию о количестве редактируемых файлов, например:

```
2 files to edit
```

После окончания редактирования следует сохранить буфер в файле file1:

```
:w
```

Редактор выведет информацию о файле: имя, а также количество находящихся в нем строк и символов. Затем можно приступить к редактированию следующего файла (в нашем примере file2), задав команду:

```
:n
```

После этого в нижней строке экрана появится информация о следующем файле, который будет редактироваться.

В некоторых случаях нужно просмотреть содержимое файла, не внося в него никаких изменений. Это можно сделать при помощи команды

```
view имя_файла
```

При этом никакие изменения в файл не вносятся.

На этом обзор редактора vi можно закончить. Более подробную информацию о редакторе vi можно получить из источников в Интернете или man-страниц.

6.2. Редактор *gedit*

Популярный текстовый редактор `gedit` разработан для обработки текстовой информации в графической оболочке GNOME. Несмотря на свою простоту, он очень удобен и предоставляет довольно широкие возможности по работе с текстовыми документами. Кроме того, `gedit` может работать в программных конвейерах, что применяется при разработке приложений для UNIX. Как и в большинстве современных редакторов, в нем используется выделение цветом при работе с различными форматами файлов (C, Perl и др.).

Начать работу с редактором `gedit` можно одним из двух способов:

- выбрать в меню соответствующие опции. Например, в операционной системе Solaris следует выбрать **Launch | Applications | Accessories | Text Editor**;
- ввести в командной строке команду `gedit`, при необходимости указав путевые имена файлов. Например, для открытия файлов 1.pl и 3.pl, находящихся на рабочем столе, нужно ввести команду

```
# gedit /Desktop/1.pl /Desktop/3.pl
```

При успешном запуске редактора `gedit` на экране появится окно (рис. 6.2).

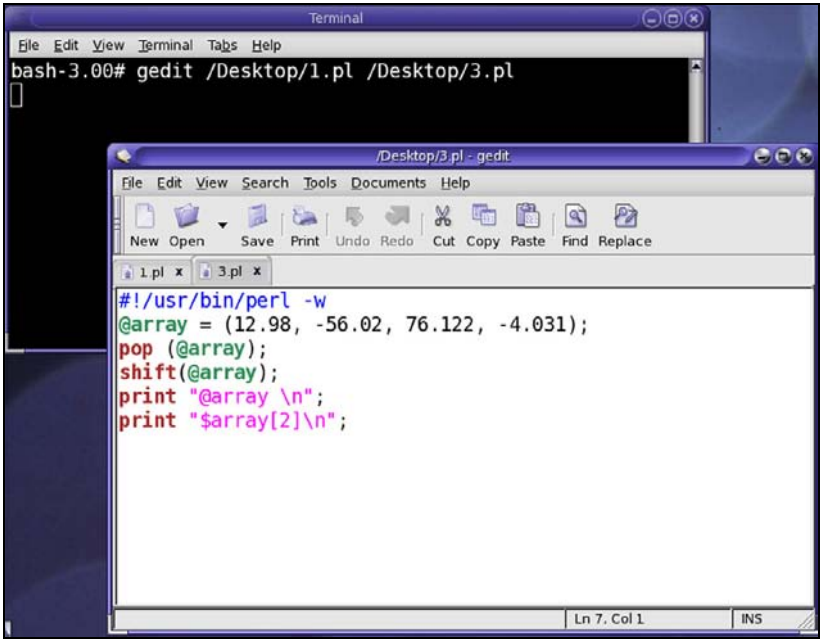


Рис. 6.2. Окно редактора `gedit` при запуске

Окно редактора `gedit` (см. рис. 6.2) содержит:

- панель основного меню, из которого можно выбрать команды для работы с файлами и данными;
- панель инструментов, содержащую часть команд, доступных из основного меню;
- рабочую область, которая может содержать несколько секций (если открыты несколько файлов одновременно). При этом каждая секция имеет заголовок, указывающий на имя файла. В данном случае редактор `gedit` открывает две секции для файлов 1.pl и 3.pl. Чтобы отобразить содержимое файла, нужно щелкнуть на заголовке соответствующей секции, а закрыть секцию можно, нажав на символ X в заголовке секции;
- строку статуса, отображающую информацию о текущем состоянии редактора `gedit`, а также вспомогательную информацию, касающуюся пунктов меню. Кроме этого, строка статуса отображает текущую позицию курсора (строку и столбец), а также режим редактирования (**INS** — при вводе, **OVR** — при перезаписи). Изменить режим редактирования можно, нажав клавишу <Insert>.

Если нажать правую кнопку мыши в окне редактора, то появится всплывающее меню, из которого можно выбрать наиболее часто используемые команды редактирования.

Редактор `gedit` позволяет выполнить одну и ту же операцию несколькими способами, что очень удобно для пользователя. Например, открыть файл для редактирования можно, используя один из нижеперечисленных вариантов:

- перетащить файл в окно редактора `gedit` из окна другого приложения, например, менеджера файлов;
- выбрать опцию меню **File | Open**;
- щелкнуть мышью на кнопке **Open** панели инструментов;
- если открыто окно менеджера файлов, то можно щелкнуть правой кнопкой мыши на файле, а затем в выпадающем меню выбрать опцию **Open With** (Открыть с помощью).

Возможности редактора `gedit` в общем не отличаются от аналогичных, предоставляемых другими редакторами. Тем не менее, `gedit` предоставляет и ряд дополнительных возможностей, улучшающих его функциональность.

К одной из них относится возможность работы редактора в программном конвейере, что позволяет принимать вывод от других программ. Например, пользователю нужно получить информацию о распределении дискового пространства для накопителей с файлами устройств /dev/... при помощи команды

`df -k`, отредактировать ее для отчета и сохранить в файле df-k.LOG. В этом случае в окне консоли можно ввести команду

```
# df -k | grep dev | gedit df-k.LOG
```

Результат выполнения команды показан на рис. 6.3.

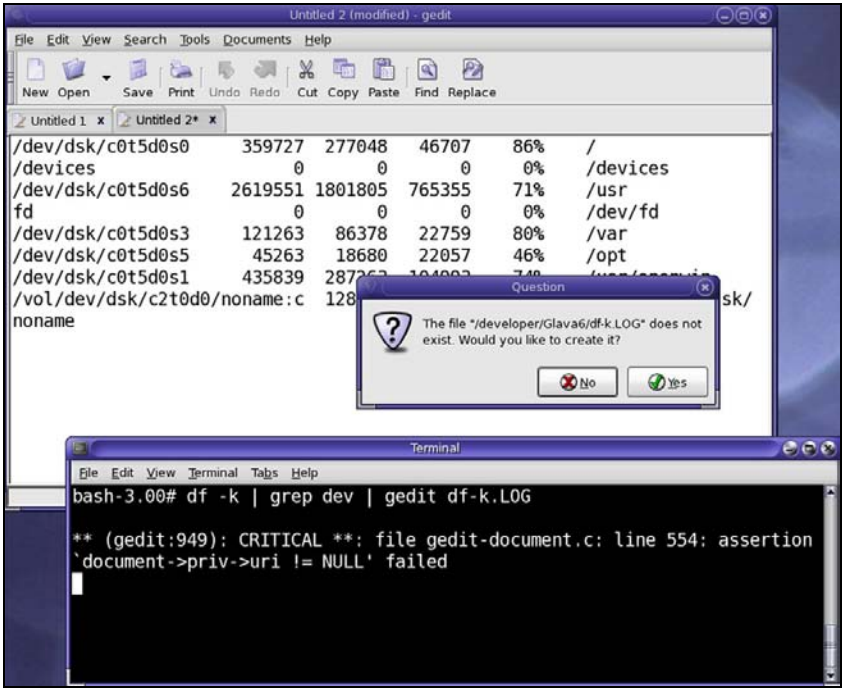


Рис. 6.3. Демонстрация работы конвейера команд с `gedit`

Поскольку файл df-k.LOG не существует, то выводится запрос на его создание. Кроме того, после выполнения команды на экран консоли выводится статусная информация.

Среди дополнительных возможностей редактора `gedit` следует выделить использование плагинов — дополнительных программ, расширяющих функциональность редактора. Плагины позволяют выполнить целый ряд функций, результаты работы которых используются в редактируемом документе. Вот перечень плагинов, которые могут использоваться редактором `gedit`:

- Document Statistics — позволяет подсчитывать количество строк, слов, символов с учетом пробелов, символов без учета пробелов и байтов. Этот плагин отображает результат в диалоговом окне;

- Indent lines — делает отступы для указанных строк или удаляет отступы для указанных строк;
- Insert Date/Time — вставляет значение текущей даты и времени в файл;
- Shell command — отображает результат выполнения команды интерпретатора shell в отдельном окне. Полученная текстовая информация может использоваться для редактирования;
- Sort — сортирует выделенный текст;
- Spell checker — выполняет орфографический контроль выделенного текста. Пользователь может сконфигурировать редактор `gedit` для автоматической или ручной проверки орфографии;
- User name — вставляет имя текущего пользователя в файл.

Для установки плагина следует выбрать опцию **Edit | Preferences**, затем выбрать вкладку **Plugins**, в которой отображается таблица, состоящая из двух колонок:

- **Enabled** — находится слева и представляет собой группу кнопок-флажков, позволяющих разрешить или отменить установку тех или иных плагинов;
- **Plugin** — отображает список плагинов, доступных в `gedit`.

Для выбора плагина следует установить флажок **Enabled** напротив выбранной программы.

Когда плагин установлен, редактор разрешает доступ к нему через определенные опции, которые `gedit` добавляет в меню (табл. 6.1).

Таблица. 6.1. *Опции выбора плагинов*

Имя плагина	Добавленный пункт меню
Document Statistics	**Tools | Document Statistics**
Indent lines	**Edit | Indent** **Edit | Unindent**
Insert Date/Time	**Edit | Insert Date and Time**
Shell command	**Tools | Run Command**
Sort	**Edit | Sort**
Spell checker	**Tools | Check Spelling** **Tools | Autocheck Spelling** **Tools | Set Language**
User name	**Edit | Insert User Name**

Рассмотрим практические примеры использования плагинов при обработке текста на примере установки программ Shell command и Document Statistics. Начнем с Document Statistics.

Установка данного плагина выполняется так, как описано ранее и показано на рис. 6.4.

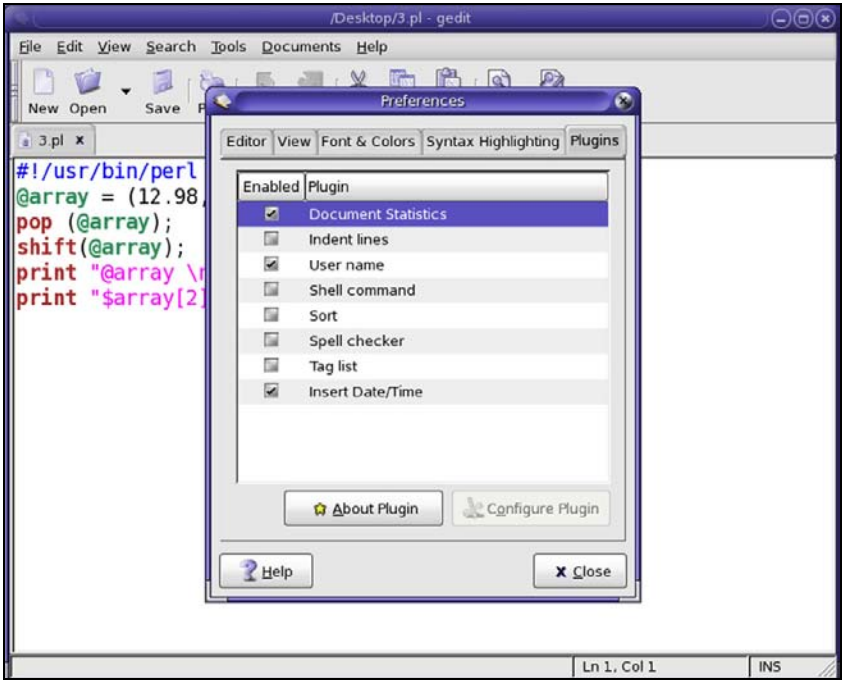


Рис. 6.4. Установка плагина Document Statistics

После этого используем только что установленный плагин для получения статистической информации об открытом в данный момент файле с именем 3.pl (рис. 6.5).

Плагин Shell command позволяет перенаправить вывод результата одной из команд интерпретатора shell в редактор `gedit`, отображая вывод в отдельном окне. Кроме того, можно скопировать выводимый командой текст в буфер, а затем вставить его в редактируемый файл. Для того чтобы можно было использовать этот плагин, следует вначале установить его, что выполняется стандартным способом, который мы рассмотрели ранее.

Далее следует выполнить последовательно несколько шагов: сначала выбрать опцию **Tools | Run Command**. Затем ввести нужную команду интерпретатора shell в поле **Command** всплывающего диалогового окна.

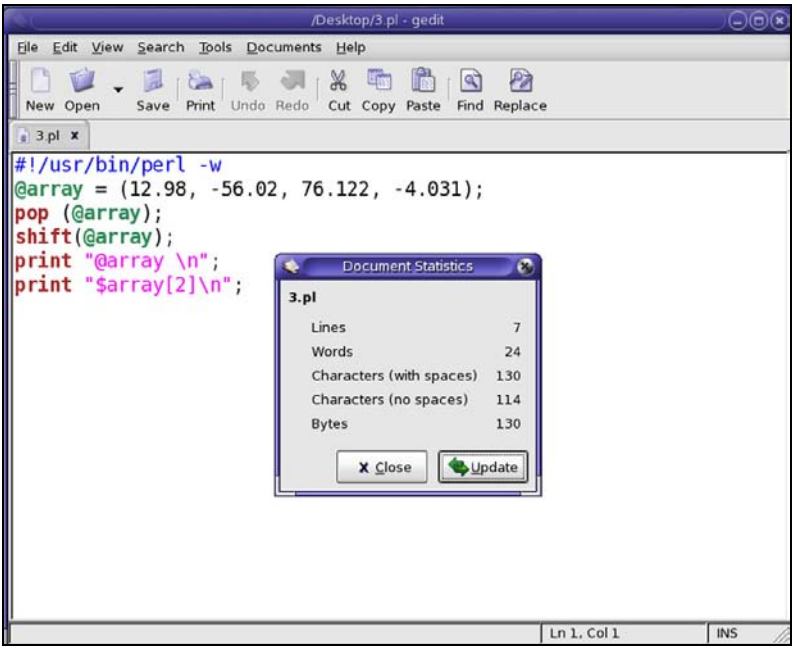


Рис. 6.5. Вывод статистики для файла 3.pl

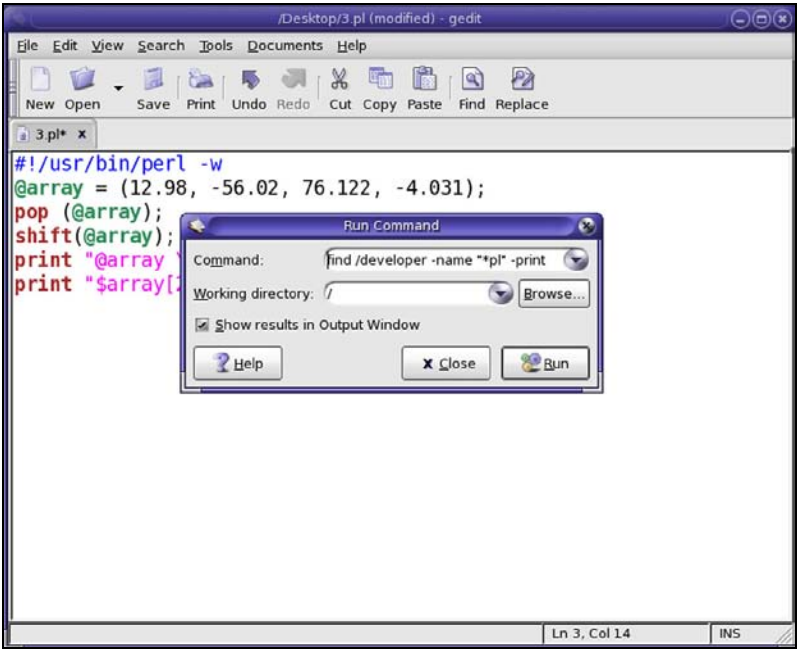


Рис. 6.6. Ввод команды в диалоговом окне для запуска плагина Shell command

Следующие два рисунка демонстрируют использование данного плагина. На рис. 6.6 показано диалоговое окно для ввода команды (в данном случае будет выполнен поиск всех файлов, оканчивающихся на pl в каталоге /developer).

Далее следует щелкнуть мышью на кнопке **Run** для запуска команды. Результат работы самой команды будет таким, как показано на рис. 6.7.

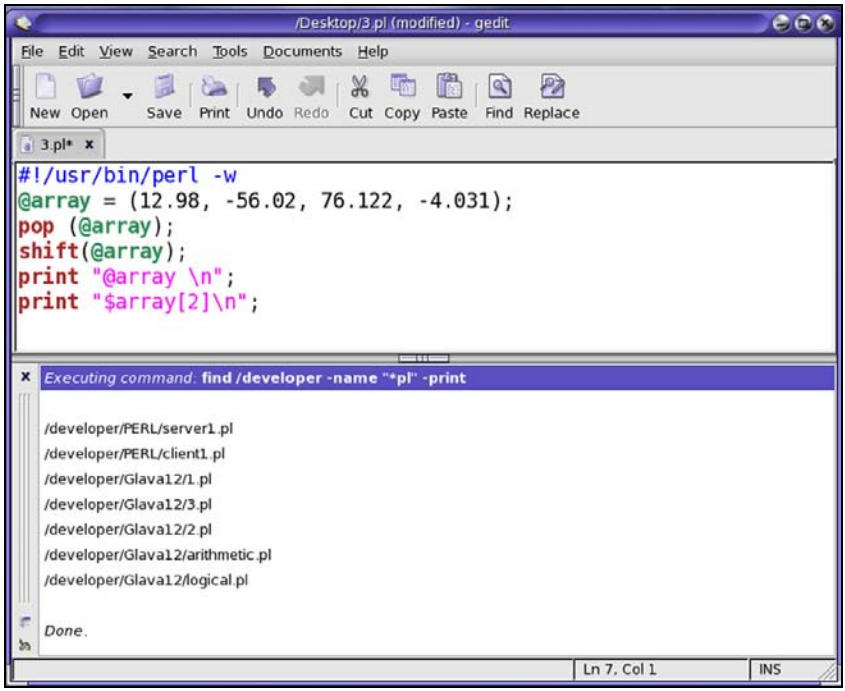


Рис. 6.7. Вывод результата выполнения команды в `gedit`

После этого результат вывода команды можно использовать при редактировании. Вначале следует выбрать фрагмент текста, используя клавишу <Shift>, после чего скопировать его в буфер памяти (команда **Copy** редактора). Далее выбрать позицию в редактируемом файле, куда нужно вставить текст, и выполнить команду **Edit | Paste**. После этого можно закрыть окно вывода, нажав символ X или выбрав в меню команду **Close**.

При запуске команд можно использовать следующие специальные символы:

- %f — используется для указания имени файла, который в данный момент обрабатывается, включая полный путь к нему;

- %n — используется для указания имени файла, который в данный момент обрабатывается, исключая полный путь к нему. В этом случае `gedit` ищет файл в рабочем каталоге.

До сих пор мы рассматривали функционирование текстового редактора `gedit` как отдельного приложения. Тем не менее, во многих случаях эту программу (как и многие другие текстовые редакторы) можно использовать для динамического формирования отчетов для других приложений, выполняющих, например, функции мониторинга ресурсов операционной системы или сбора и анализа данных, поступающих из нескольких источников в сети. При этом основное приложение играет роль предварительного обработчика (фильтра) данных, передавая их редактору `gedit` для окончательной обработки посредством именованного или неименованного канала. Напомню, что `gedit` может функционировать в конвейере команд, а все языки высокого уровня, например, C и Perl, включают в себя функции для работы с программными каналами.

Хочу заметить, что применение `gedit` вместе с приложениями на языках высокого уровня позволяет реализовать весьма изощренные алгоритмы обработки данных и получить результаты в удобной форме.

Рассмотрим пример использования редактора `gedit` в программе на языке Perl. Сразу же оговорюсь, что основы разработки программ на Perl детально будут рассмотрены в *главе 12*, здесь же я хочу продемонстрировать принципы взаимодействия приложения и текстового редактора.

Предположим, что в системе имеется текстовый файл с именем myfile, содержимое которого показано далее:

```
# cat myfile
1-st String
2-nd String
3-rd String
First String
Second String
Third String
```

В данном файле необходимо выделить строки, не содержащие символов цифр, и передать их для обработки в редактор `gedit`. Реализация такого алгоритма выполнена в программе gedit_ipc.pl, исходный текст которой показан далее:

```
#!/usr/bin/perl -w
$myfile = shift or die "Usage: $0 file\n";
open(FD, "$myfile") or die "open:$!";
```

```
@data = <FD>;
close(FD);
open(GED, "|gedit") or die "open: $!";
foreach (@data)
 {
  next if ($_ =~ !/^.*[0-9].*$/);
  print GED $_;
 }
close(GED);
```

Программа принимает в качестве единственного аргумента имя файла. В данном случае для обработки файла myfile нужно выполнить команду

```
# ./gedit_ipc.pl myfile
```

В результате выполнения программы строки

```
First String
Second String
Third String
```

из файла myfile будут переданы в открывшееся окно текстового редактора `gedit`.

Я не буду анализировать алгоритм работы данной программы и смысл операторов — читатели без особого труда смогут сделать это самостоятельно, ознакомившись с основами разработки на языке Perl в *главе 12*. Хочу лишь обратить внимание на то, как используется программа `gedit`.

Оператор

```
open(GED, "|gedit") or die "open: $!";
```

реализует механизм программного конвейера, в котором редактор `gedit` принимает входные данные от приложения. Приведенный пример является простейшим, тем не менее, на его основе можно разработать намного более сложные алгоритмы обработки данных.

Более подробную информацию о возможностях текстового редактора `gedit`, а также новые версии этой программы можно обнаружить в многочисленных источниках в Интернете.

6.3. Редактор *Kate*

Приступим к изучению текстового редактора `Kate` — одного из самых распространенных и самых мощных текстовых редакторов для оболочки KDE. Здесь будут рассмотрены особенности функционирования программы, начиная с версий 2.2 и выше. Кроме общих для всех текстовых редакторов с гра-

фическим интерфейсом принципов функционирования в Kate реализованы некоторые дополнительные возможности:

- осуществляется подсветка и выделение цветом редактируемого текста файлов с исходными текстами программ на C/C++, Perl, Java, а также поддержка HTML-страниц;
- реализован многодокументный интерфейс пользователя;
- предусмотрено функционирование электронной почты;
- имеется встроенный эмулятор терминала;
- реализовано выполнение внутренних команд редактора посредством встроенной командной строки;
- обеспечивается повышенная функциональность за счет использования дополнительных программ (плагинов).

Кроме того, редактор Kate позволяет создавать многодокументные проекты, что особенно удобно для разработчиков программного обеспечения.

6.3.1. Запуск редактора *Kate*

Запуск редактора осуществляется либо через меню, либо из командной строки. Запустить из меню редактор Kate можно из графической оболочки KDE, для чего следует щелкнуть мышью на пиктограмме в виде литеры K в левом нижнем углу экрана. Далее в развернутом меню следует выбрать опции **Utilities | Editors**, после чего из списка программ-редакторов выбрать **Kate** (или **kate**). Если при работе с редактором не было создано ни одной сессии, то запускается сессия по умолчанию (**Default Session**).

Термин "сессия" используется для обозначения той или иной рабочей конфигурации, с которой работает редактор Kate. Например, сессия может включать один или несколько файлов для обработки, специфичные настройки параметров графического интерфейса и т. д.

Запуск редактора Kate иллюстрируется рис. 6.8.

Обратите внимание, что при открытии сессии по умолчанию редактор Kate не открывает никаких существующих файлов. Далее можно выбрать пункт меню **Open Session** и продолжить работу настройками по умолчанию, либо выбрать пункт **New Session** и настроить свою рабочую среду в Kate. Персональные настройки можно сохранить и в процессе работы с **Default Session**, как это показано на рис. 6.9.

Для создания сессии нужно выбрать опцию **Sessions | Save As...**, после чего в открывающемся диалоговом окне ввести имя сессии и нажать кнопку **OK**.

В данной сессии будут сохранены текущие настройки редактора вместе с именами обрабатываемых в настоящий момент файлов (в данном случае tcpserver1.pl).

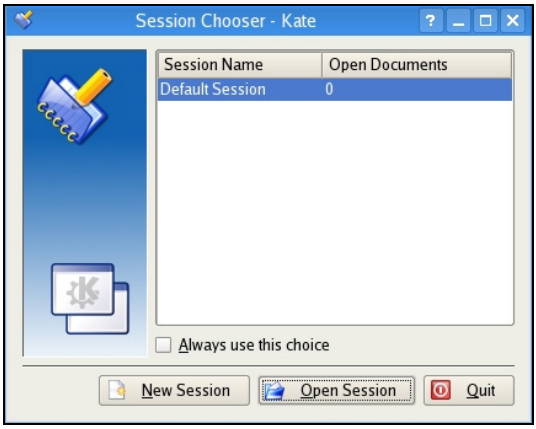


Рис. 6.8. Запуск редактора `Kate`

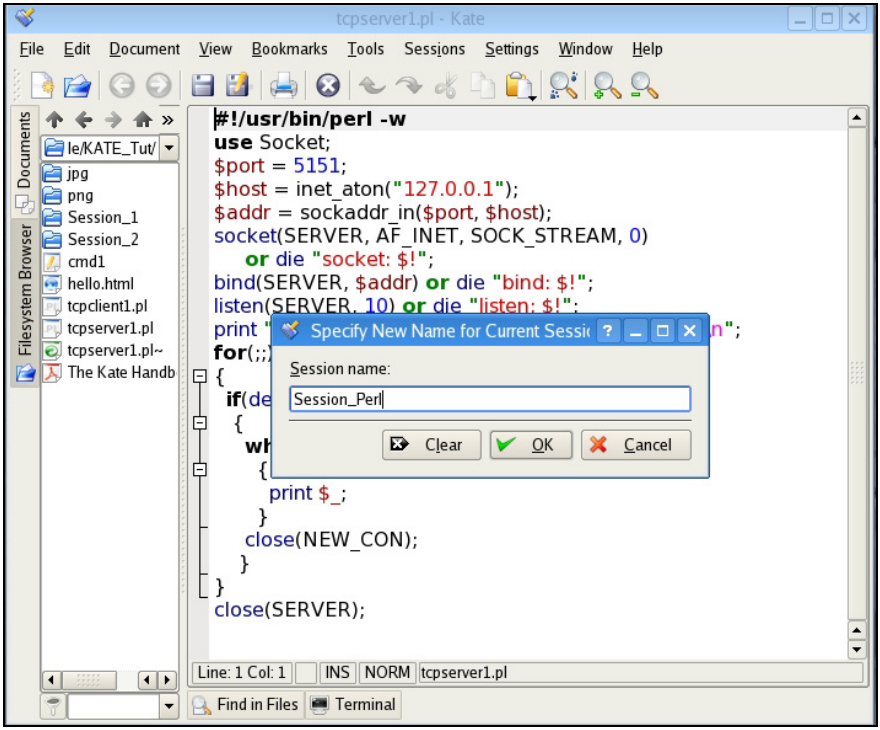


Рис. 6.9. Создание сессии Session_Perl на основе текущих настроек

При следующем запуске редактора Kate в диалоговом окне выбора сессии будет присутствовать список из двух сессий (рис. 6.10).

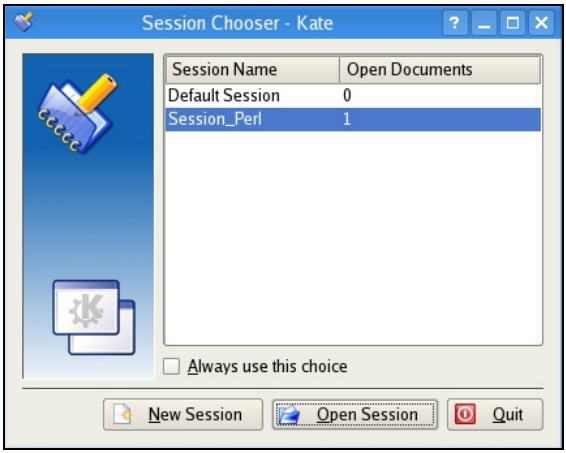


Рис. 6.10. Диалоговое окно выбора сессии

Обратите внимание на поле **Open Documents** — здесь указано, что при открытии сессии Session_Perl будет открыт один документ (в данном случае файл tspserver1.pl). В любой момент во время работы можно переименовать или удалить сессию, выбрав опцию **Sessions | Manage...** из меню (рис. 6.11).

Редактор Kate можно запускать и из командной строки, с параметрами или без них. Ввиду ограниченности объема рассмотрим только некоторые важные команды для запуска Kate.

Команда

```
# Kate файл
```

позволяет открыть или создать файл с именем *файл*. При этом в командной строке можно указать и удаленный файл (если имеется доступ к Интернету), тогда он будет загружен для обработки, например:

```
# Kate ftp://ftp.kde.org/pub/kde/README_FIRST
```

Следующая команда

```
# Kate --help
```

отображает основные опции командной строки. Команда

```
Kate --help-kde
```

выводит на экран опции, которые используются для настроек интерфейса между редактором Kate и оболочкой KDE.

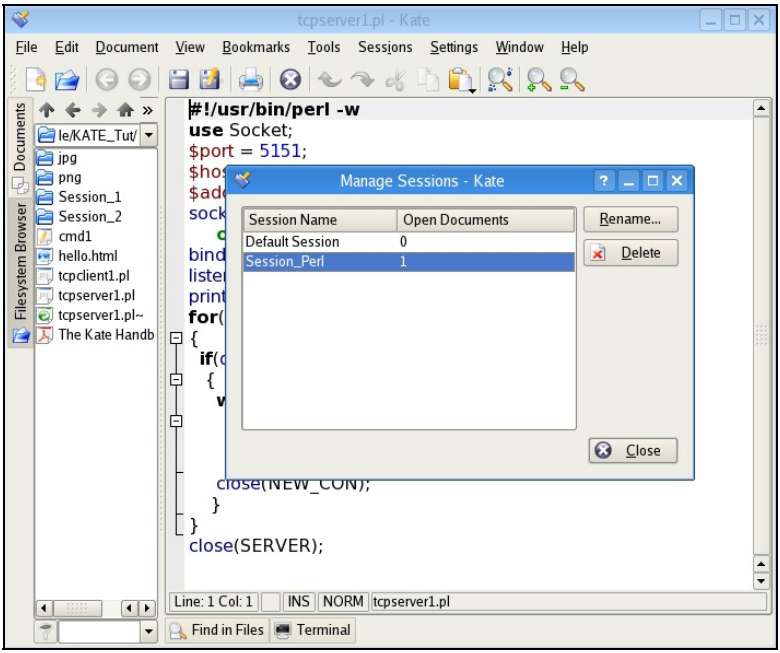


Рис. 6.11. Диалоговое окно управления сессиями

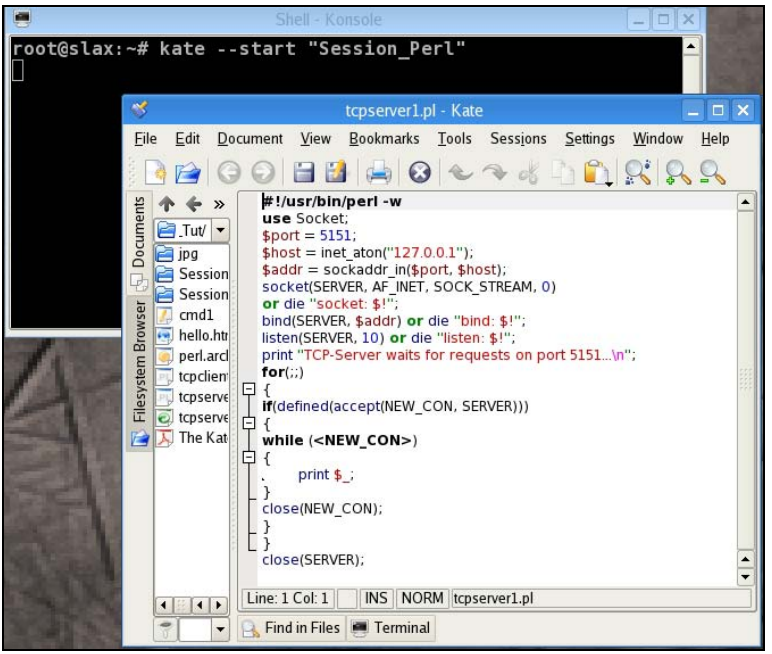


Рис. 6.12. Запуск сессии Session_Perl из командной строки

Запуск сессии можно выполнить при помощи следующей командной строки:

```
Kate --start имя
```

Здесь программа `Kate` запускает сессию с именем *имя*. Если сессия не существует, она создается. Если вместо параметра *имя* использовать последовательность символов "", то запускается сессия по умолчанию (Default Session). Например, для запуска сессии Session_Perl командная строка будет выглядеть так:

```
# Kate --start "Session_Perl"
```

Результат выполнения этой команды показан на рис. 6.12.

Обратите внимание, что при открытии сессии из командной строки диалоговое окно выбора сессии не появляется.

Нередко требуется получить данные для редактирования из стандартного ввода (stdin). В этом случае командная строка будет такой:

```
Kate --stdin
```

По окончанию ввода данных из окна консоли нужно нажать комбинацию клавиш <Ctrl>+<d>, при этом данные передаются в буфер памяти редактора `Kate` и отображаются в окне редактирования. Пример выполнения этой команды иллюстрирует рис. 6.13.

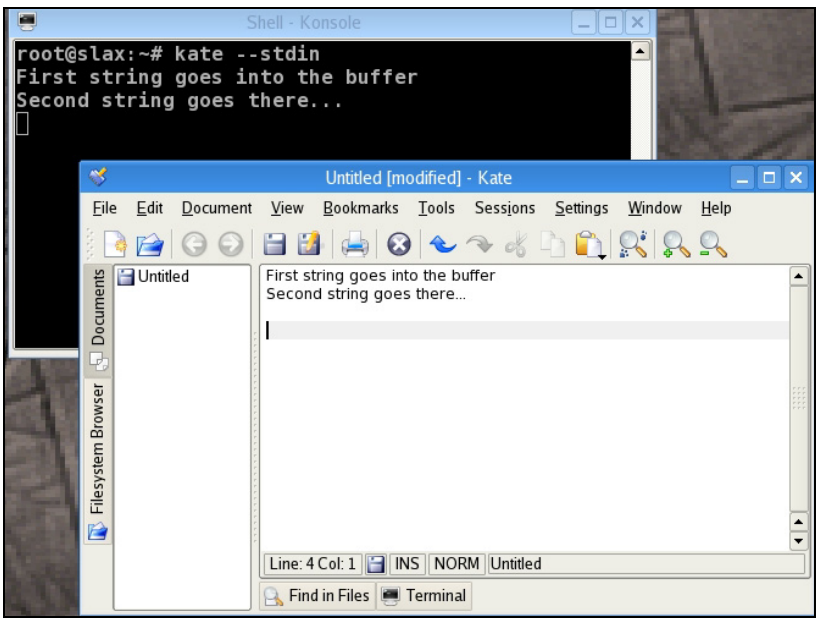


Рис. 6.13. Прием данных со стандартного ввода

6.3.2. Работа в редакторе

В этом разделе мы рассмотрим особенности работы с редактором `Kate`. Хочу сразу заметить, что общие принципы работы с данными (открытие, редактирование и сохранение файлов, работа с буфером обмена и т. д.) анализироваться не будут, поскольку они реализованы во всех текстовых редакторах с графическим интерфейсом и хорошо описаны в литературе. Мы будем рассматривать только те особенности функционирования редактора `Kate`, которые делают его действительно мощным инструментом обработки текста с очень широкими дополнительными возможностями.

В редактор `Kate` включена развитая справочная система — для вызова справки достаточно нажать клавишу <F1> или использовать навигацию по меню **Help Contents**.

После первого запуска редактора `Kate` раскрываются два окна. Поверх них размещается панель инструментов с пиктограммами, над которой находится панель основного меню. Левое окно представляет собой комбинацию двух боковых окон — **Documents** и **Filesystem Browser**. Переключения между окнами осуществляются посредством щелчков мыши на соответствующих вкладках.

В том случае, если при запуске `Kate` указано имя файла, его содержимое будет отображаться в правом окне, а имя — в окне **Documents**. Окно **Filesystem Browser** используется для открытия нужного файла. Включить/отключить окна **Documents** и **Filesystem Browser** можно из меню **Window | Tool Views**.

Файлы можно открывать не только классическим способом, но и используя технологию перетаскивания объектов (Drag and Drop), поддерживаемую оболочкой KDE. При этом нужный файл перетаскивается мышью на правое окно редактора, что приводит к автоматическому открытию файла. Следует сказать, что не имеет значения, откуда перетаскивается файл: из окна менеджера файлов, FTP-сайта или браузера.

Расположение окон редактора `Kate`, описанное выше, является принятым по умолчанию. Тем не менее, расположение окон **Documents** и **Filesystem Browser**, а также окна **Terminal** эмулятора терминала можно изменять. Для этого нужно щелкнуть правой кнопкой мыши на вкладке соответствующего окна, затем из выпадающего меню выбрать одну из опций:

- **Right Sidebar** — разместить справа;
- **Top Sidebar** — разместить вверху;
- **Bottom Sidebar** — разместить внизу.

Как будут размещены окна **Documents** и **Filesystem Browser** при выборе опции **Bottom Sidebar**, показано на рис. 6.14.

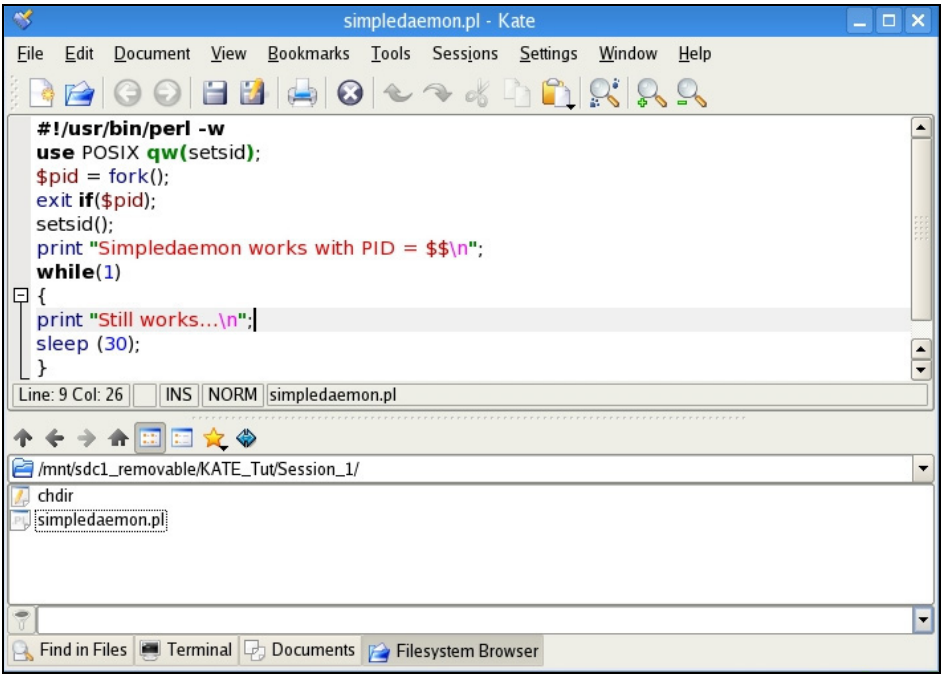


Рис. 6.14. Нестандартное расположение окон

Окна **Documents**, **Filesystem Browser** и **Terminal** можно скрывать и отображать посредством опций **Hide Sidebars**, **Hide Filesystem Browser**, **Show Documents**, **Show Terminal** меню **Window**.

Редактор `Kate` может обрабатывать несколько документов одновременно, для чего предусмотрена возможность разделения области редактирования на несколько окон (фреймов). Это позволяет в одно и то же время работать как с разными документами, так и с копиями одного и того же документа. Когда один и тот же документ доступен в двух окнах редактирования, то любые изменения, произведенные в одном фрейме, немедленно отображаются в другом. В настоящее время редактор `Kate` не позволяет редактировать один и тот же документ в разных окнах независимо друг от друга.

Имеется несколько вариантов одновременной работы с несколькими документами. Можно работать с несколькими документами в одном и том же окне редактирования — для этого следует создать новую вкладку (new tab). Это можно выполнить посредством команды **New Tab** меню **Window**. Вот как будет выглядеть область редактирования для работы с двумя вкладками (рис. 6.15).

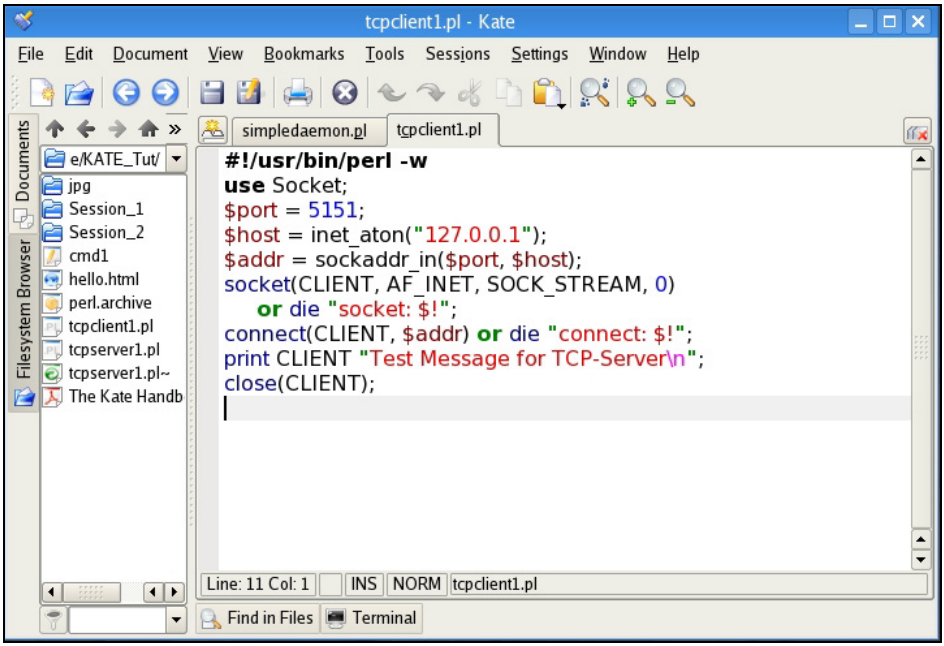


Рис. 6.15. Окно редактирования с двумя вкладками

Как видно из рисунка, созданы вкладки для файлов simpledaemon.pl и tcpclient1.pl. Для перехода к редактированию, например, файла simpledaemon.pl следует щелкнуть на соответствующей вкладке. В общем случае для перехода на соседние вкладки можно воспользоваться опциями **Activate Next Tab** (Активизировать следующую вкладку) и **Activate Previous Tab** (Активизировать предыдущую вкладку) меню **Window**. Хочу обратить ваше внимание на то, что при таком способе работы с несколькими документами можно просматривать и редактировать фактически один файл, хотя можно очень быстро переключиться на другой документ. Закрыть текущую (рабочую) вкладку можно из того же меню **Window**, выбрав опцию **Close Current Tab**. Здесь и везде далее по тексту под рабочим или текущим окном (вкладка также является одной из разновидностей окна) подразумевается объект, которому принадлежит фокус.

Для работы с несколькими документами одновременно можно использовать и другой способ — разделить область редактирования окна на две или более частей по горизонтали или по вертикали. Это позволяет просматривать одновременно несколько документов, хотя делает менее удобным процесс редактирования из-за разделения одной области редактирования на несколько. Для разделения области редактирования по вертикали или горизонтали, соответ-

ственно, используются опции **Split Vertical** и **Split Horizontal** меню **Window**. Закрыть текущее (рабочее) окно можно при помощи опции **Close Current Window** меню **Window**.

Пример работы с разделенными окнами показан на рис. 6.16.

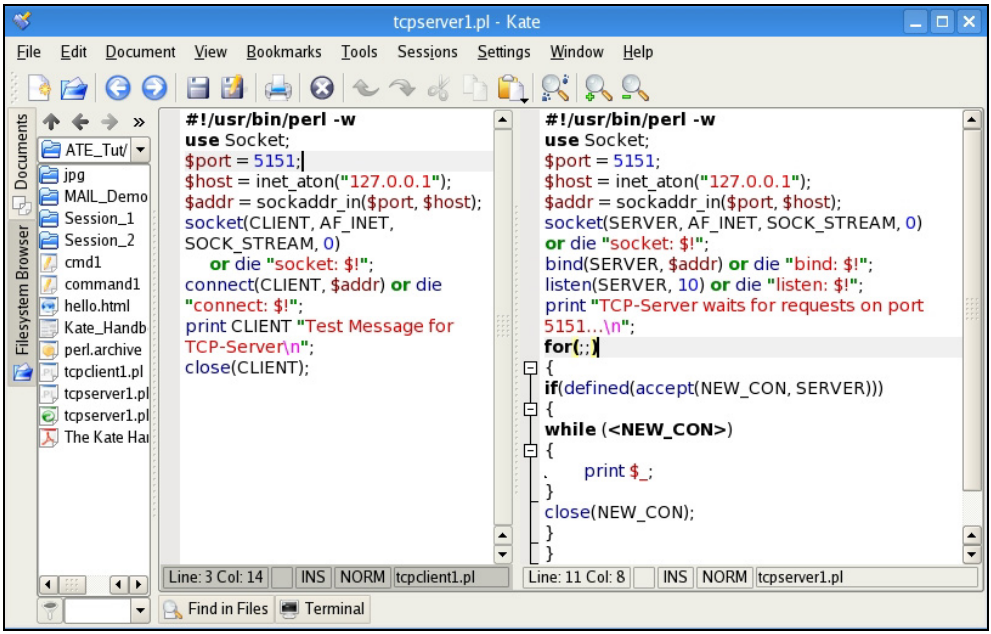


Рис. 6.16. Работа в разделенных по вертикали окнах

Наконец, еще один метод для одновременной обработки документов состоит в создании нового окна (фактически это копия выполняющейся программы `Kate`). При этом копия программы работает независимо от оригинала; более того, основную программу можно закрыть без каких-либо последствий для вновь созданной. Создать новое окно можно при помощи опции **New Window** меню **Window**.

Расширенные возможности *Kate*

Остановимся на расширенных возможностях редактора `Kate`, улучшающих его функциональные возможности. Во-первых, `Kate` позволяет отправлять электронную почту непосредственно из рабочей среды, для чего следует выбрать опцию **File | Mail...** (рис. 6.17).

Следует отметить, что для функционирования электронной почты в `Kate` необходимо иметь проинсталлированную программу KMail.

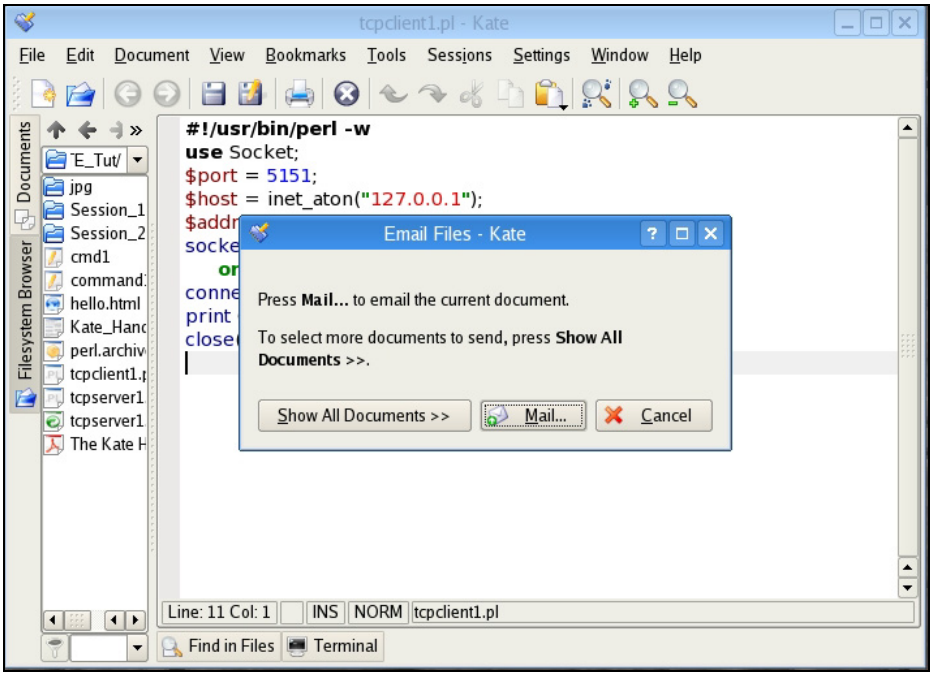


Рис. 6.17. Отправка почты из редактора `Kate`

Следующая особенность редактора `Kate` — наличие встроенного эмулятора терминала (built-in Terminal Emulator), представляющего собой обычное консольное приложение, имеющееся также в оболочке KDE в виде отдельной программы. Встроенный эмулятор позволяет запускать приложения и выполнять команды интерпретатора shell непосредственно в редакторе, что само по себе очень удобно. Кроме того, что еще важнее, результаты работы программ, запущенных в окне эмулятора терминала, можно передавать для дальнейшей обработки редактору посредством буфера обмена. Наконец, эмулятору можно передавать из окна редактирования введенные данные, которые могут интерпретироваться как команды и выполняться обычным образом.

Окно эмулятора терминала по умолчанию располагается внизу, ниже области редактирования программы `Kate`. Если по каким-то причинам окно скрыто, то можно отобразить его, используя опцию **Tool Views | Show Terminal** меню **Window**.

Установить параметры (тип и размер шрифта, цветовую схему и т. д.) эмулятора терминала можно, щелкнув правой кнопкой мыши на окне эмулятора и выбрав нужную опцию выпадающего меню.

В практическом плане результат выполнения команды, запущенной в эмуляторе терминала, передать в окно редактирования довольно просто. Для этого нужно выполнить два шага:

1. Выделить содержимое требуемой области в окне терминала, щелкнуть правой кнопкой мыши на выделенной области и в выпадающем меню выполнить команду **Copy**. В результате содержимое выделенной области окажется в буфере обмена.
2. Перейти на область редактирования и, получив фокус, щелкнуть правой кнопкой мыши, после чего в выпадающем меню выполнить команду **Paste**.

Естественно, что для копирования/вставки текста можно воспользоваться и альтернативными способами — опциями основного меню или быстрыми клавишами.

Пример копирования результата выполнения команды `df -l` в область редактирования представлен на рис. 6.18.

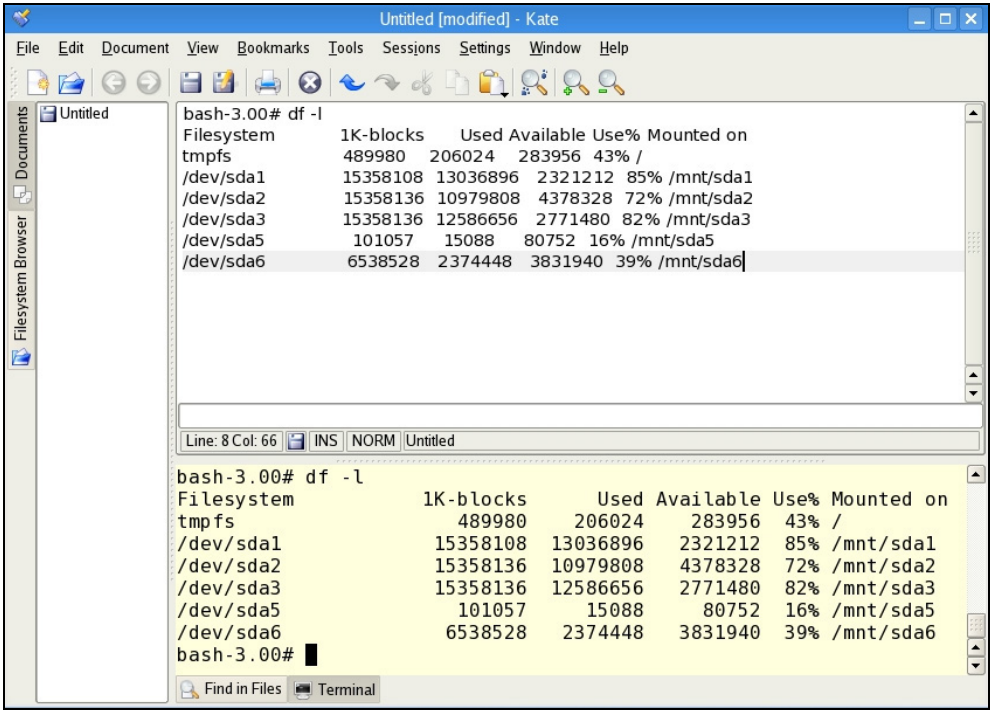


Рис. 6.18. Копирование результата выполнения команды в окно редактирования

После выполнения копирования данные можно обрабатывать обычным образом.

Следует сказать, что существует еще один способ выполнения конвейера команд, позволяющий получить подобный результат. Этот способ базируется на использовании команды

```
Kate --stdin
```

обеспечивающей прием данных со стандартного ввода. Это означает также и то, что данная команда может принимать результат выполнения любой программы, направляющей данные на стандартный вывод. Иными словами, для получения данных программа `Kate` может использовать конвейер команд. Например, результат выполнения команды поиска файлов *.pl

```
find . name "*.pl" -print | Kate -stdin
```

в каталоге /mnt/sdc1_removable может выглядеть так, как показано на рис. 6.19.

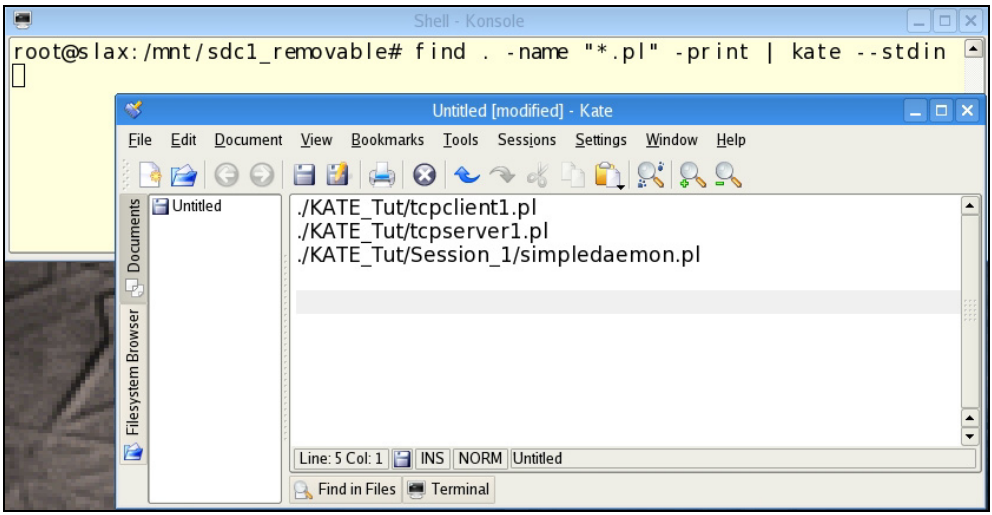


Рис. 6.19. Передача результата выполнения команды поиска редактору `Kate`

Здесь редактор `Kate` принимает данные команды `find` и помещает их в окно редактирования.

Замечательным свойством редактора `Kate` является возможность выполнять последовательность команд, введенных в области редактирования, в эмуляторе терминала. Это очень удобно при отладке командных файлов, особенно учитывая то, что можно одновременно работать с несколькими окнами. Должен заметить, что корректность записи команд лежит, естественно, на поль-

зователе, поскольку командный интерпретатор будет воспринимать полученные данные как команды.

Для выполнения команд, введенных в окне редактирования, в эмуляторе терминала нужно выбрать опцию **Tools | Pipe to Console**. Далее приводится пример выполнения команд, введенных в окне редактирования. Предположим, что в окне редактирования мы ввели две команды: `cd` и команду для архивирования файлов, оканчивающихся на *pl, а затем вызвали опцию **Tools | Pipe to Console**. После этого появится предупреждение о том, что введенный текст будет интерпретироваться как команды (рис. 6.20).

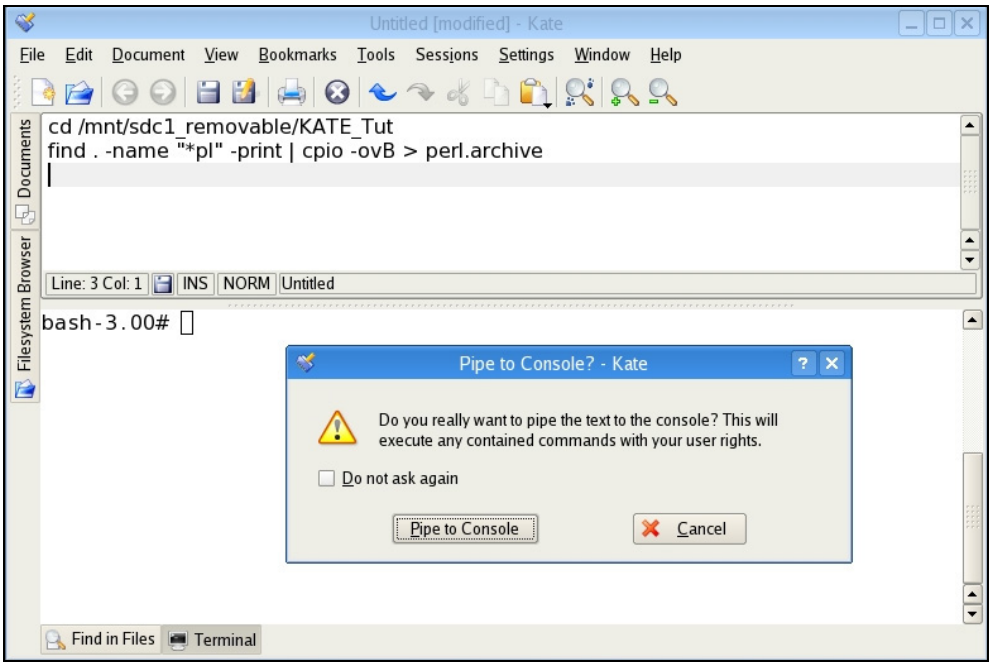


Рис. 6.20. Вызов конвейера для передачи команды эмулятору терминала

После этого нужно нажать кнопку **Pipe to Console**, если вы не сомневаетесь в том, что делаете, или отменить передачу команд, нажав кнопку **Cancel**.

После выполнения команд, введенных в окне редактирования, получим следующий результат в окне эмулятора терминала (рис. 6.21).

Комбинируя разные возможности, предоставляемые пользователю редактором `Kate`, можно создать собственную среду разработки. Приведу пример. Пусть необходимо отладить два сетевых приложения, написанные на языке Perl, причем одно из них является сервером TCP, а другое — клиентом TCP.

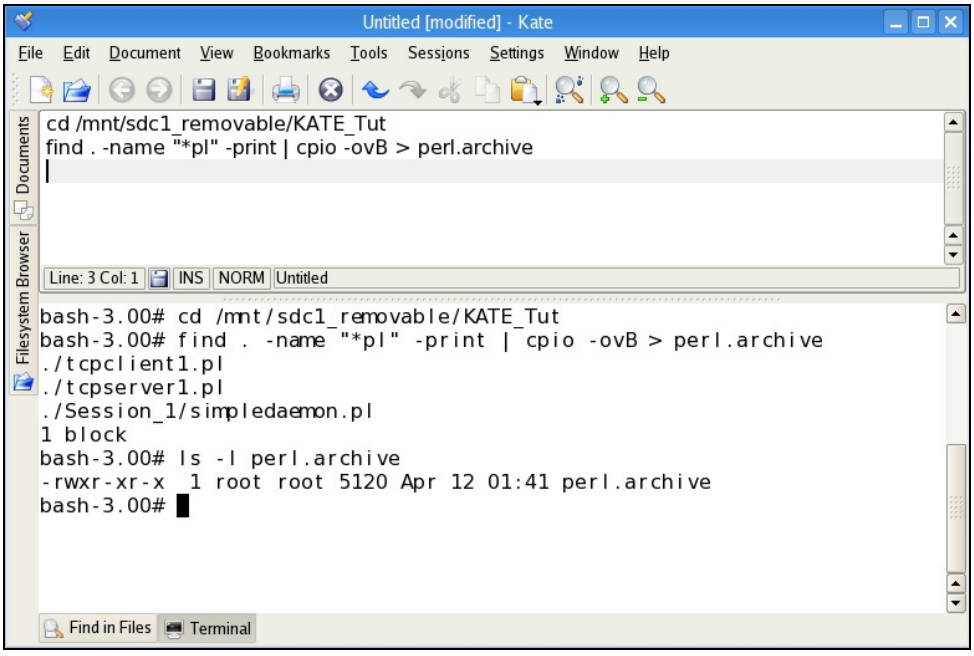


Рис. 6.21. Результат выполнения команды архивирования

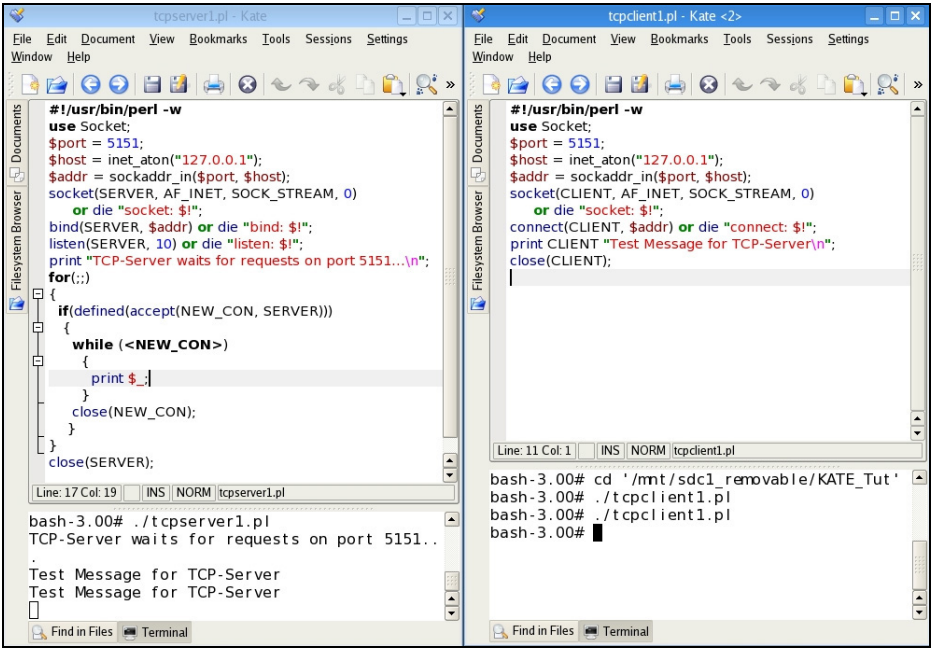


Рис. 6.22. Отладка двух приложений в редакторе `Kate`

Лучше всего в данной ситуации использовать два отдельных окна редактирования, или, что более точно, две копии редактора `Kate` — в этом случае каждая из отлаживаемых программ получает в свое распоряжение отдельный эмулятор терминала. Пример запуска программ сервера и клиента, а также результат их взаимодействия показан на рис. 6.22. Кроме возможности запуска и выполнения команд из встроенного эмулятора терминала, редактор `Kate` позволяет запускать на выполнение и внешние командные файлы. Для этого в меню **Tools** предусмотрена опция **External Tools | Run Script**. При помощи этой опции можно запускать внешние по отношению к работающему редактору приложения, которые могут использовать данные, относящиеся к обрабатываемому в данный момент документу, например, URL документа, каталог, где он находится, текст и т. д.

Пользователь сам должен установить данный инструментарий при помощи панели конфигурации **External Tools**.

Дополнительные возможности редактора *Kate*

В данном разделе мы рассмотрим некоторые дополнительные возможности по обработке данных, предоставляемые редактором `Kate`, — поиск данных в документах, использование встраиваемых программ (плагинов), а также применение внутренних команд редактора.

Для поиска текста следует вызвать окно диалога **Find Text** из опции меню **Edit | Find...**, затем ввести строку шаблона для поиска и нажать кнопку **OK**. Если поиск начинается с позиции курсора и до конца текста совпадений не обнаружено, то пользователю предлагается продолжить поиск с начала документа.

При обнаружении совпадения с шаблоном соответствующая позиция в тексте подсвечивается. Чтобы продолжить дальнейший поиск по шаблону, даже если окно **Find Text** скрыто от пользователя, можно использовать опцию **Edit | Find Next** или нажать клавишу <F3>. Если требуется выполнить дальнейший поиск в обратном направлении, следует выбрать опцию меню **Edit | Find Previous** или нажать комбинацию клавиш <Shift>+<F3>.

В качестве шаблона для поиска можно указывать как обычную строку символов, так и регулярное выражение, в котором указываются символы, имеющие специальное значение (регулярные выражения детально описаны в *главе 12*). Далее приводится пример поиска по шаблону, в качестве которого используется регулярное выражение (рис. 6.23).

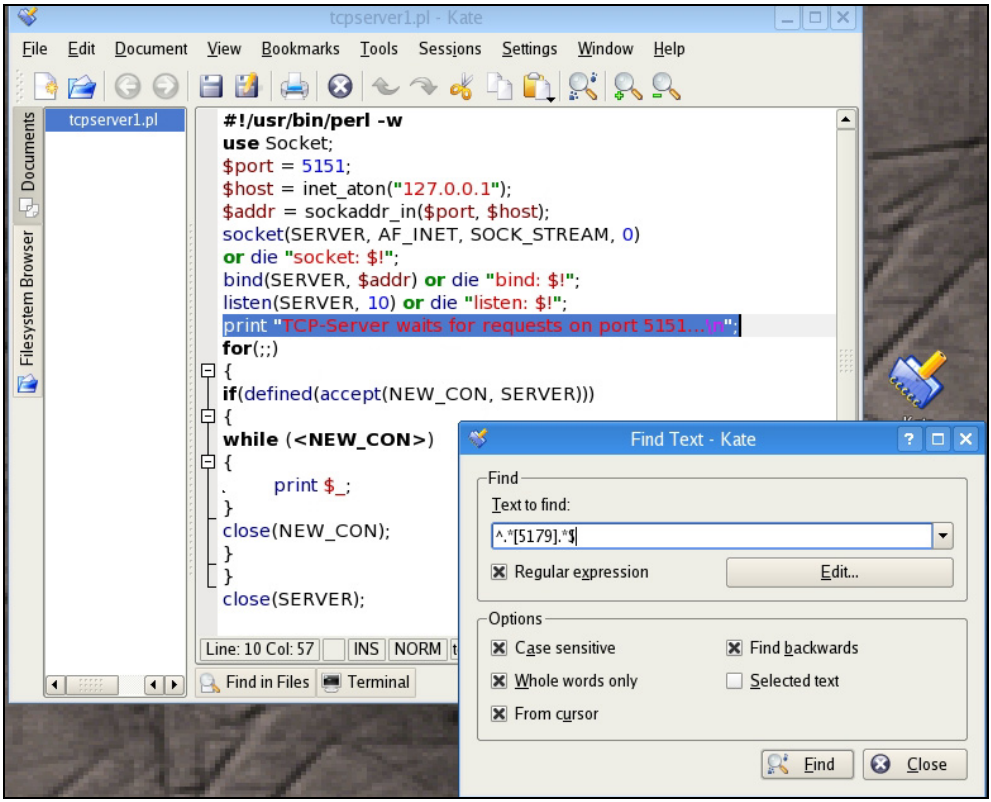


Рис. 6.23. Поиск по шаблону в открытом файле

Здесь в качестве шаблона поиска используется регулярное выражение `^.*[5179].*$`, означающее "любое выражение, которое начинается с любых символов, заканчивается любыми символами и содержит цифры 1, 5, 7, 9".

Данное выражение содержится в следующих строках:

```
$port = 5151;
$host = inet_aton("127.0.0.1");
listen(SERVER, 10) or die "listen: $!";
print "TCP-Server waits for requests on port 5151...\n";
```

На рисунке отображен момент поиска, когда данное выражение обнаружено в строке, начинающейся с "print" (выделено подсветкой).

Обратите внимание на кнопку **Edit** диалогового окна **Find Text - Kate** — она позволяет упростить выбор специальных символов для формирования выражения через более понятные пользователю обозначения.

Кроме поиска по шаблону можно выполнить и замену определенных выражений на другие. Эту операцию можно осуществить посредством опции **Edit | Replace...** или через нажатие комбинации клавиш быстрого вызова <Ctrl>+<R>. Рисунок 6.24 иллюстрирует эту операцию.

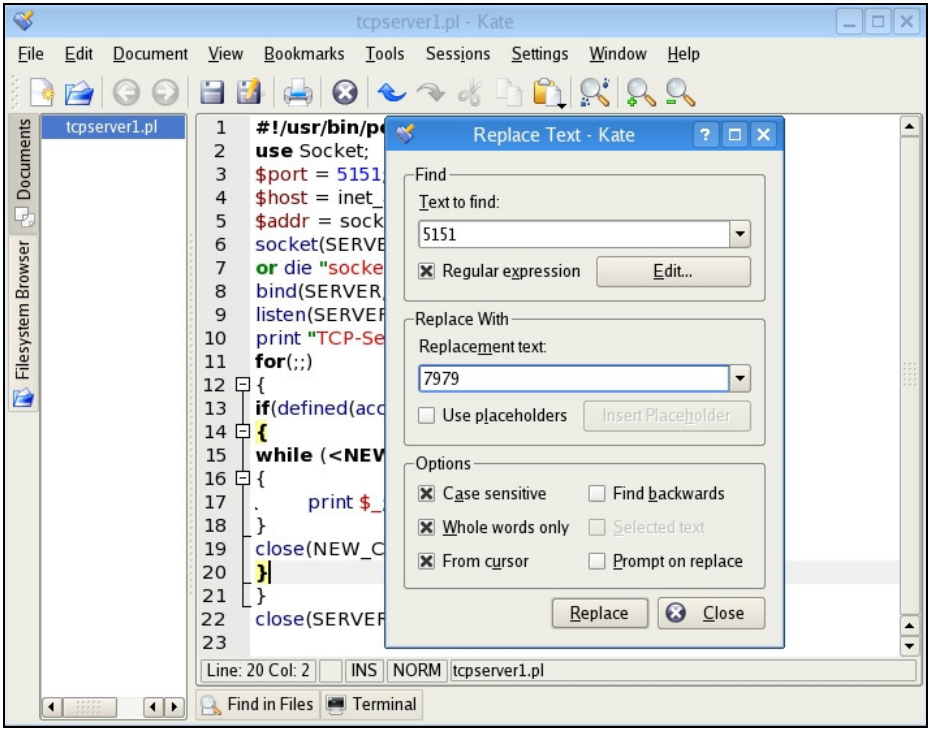


Рис. 6.24. Замена выражения в документе

При вызове команды замены в появившемся диалоговом окне **Replace Text - Kate** следует заполнить поле **Text to find:** (здесь указывается выражение или шаблон выражения для поиска) и поле **Replacement text:** (указывает, каким выражением следует заменить обнаруженное слово). В данном примере при обнаружении выражения 5151 оно будет заменено на 7979. Для начала поиска и замены следует нажать кнопку **Replace**. После выполнения операции замены будут выполнены в строках 3, 4, 9, 10 файла tcpserver1.pl.

Редактор `Kate` позволяет выполнить поиск по образцу не только в открытых документах, но и в файлах, которые в данный момент не обрабатываются. Такая возможность обеспечивается вызовом инструмента поиска **Find in Files**, расположенного внизу окна приложения. Пример выполнения такого поиска иллюстрируется на рис. 6.25.

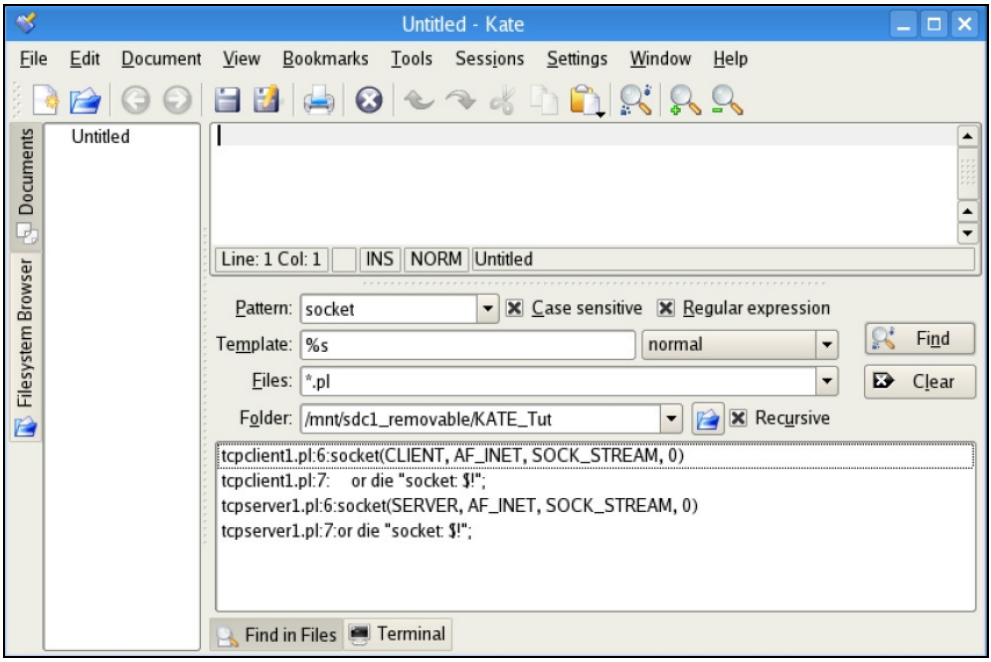


Рис. 6.25. Поиск по шаблону в файлах выбранного каталога

В простейшем варианте для выполнения поиска нужно в окне редактирования **Pattern** указать выражение, которое следует искать (в данном случае socket). В окне **File** нужно указать шаблон файла (в данном случае поиск ведется в файлах Perl, имеющих расширение pl), а в окне редактирования **Folder** — путь к каталогу для поиска, нажав пиктограмму рядом с окном и выбрав путь.

Результат выполнения поиска отображается в расположенном ниже окне с указанием имени файла и номера строки, в которой обнаружено совпадение.

Редактор `Kate` облегчает работу пользователей благодаря еще одной возможности — использованию плагинов, встраиваемых в редактор для выполнения тех или иных дополнительных функций. Для установки плагинов в редакторе нужно воспользоваться опцией **Settings | Configure Kate...**, после в диалоговом окне **Configure** выбрать опцию **Plugins**. При этом в окне справа появится список плагинов, которые можно установить.

Далее приводится пример установки плагина KTextEditor Insert File Plugin (рис. 6.26).

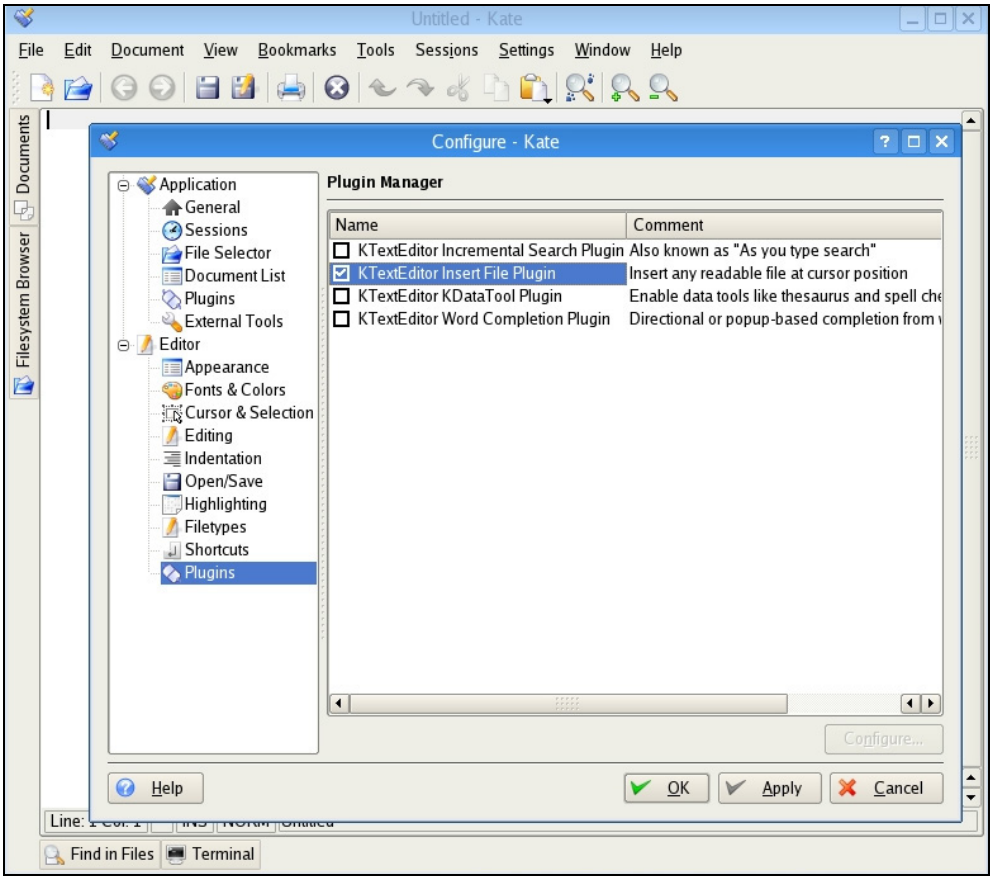


Рис. 6.26. Установка плагина в редакторе `Kate`

Этот плагин позволяет вставить содержимое выбранного файла непосредственно в редактируемый документ. После установки плагина в меню **Tools** появится опция **Insert File...**, которую и нужно использовать для вставки содержимого файла в документ.

Редактор `Kate` позволяет выполнить большинство операций с документом путем выбора той или иной опции из многочисленного списка, разворачивающегося при выборе меню. Тем не менее, в программе имеется встроенная командная строка, позволяющая выполнить все необходимые манипуляции текстом. Для вызова командной строки нужно выбрать опцию **View | Switch to Command Line**, после чего в нижней части окна редактирования появится строка редактирования, в которой можно вводить команды. Для вызова командной строки можно воспользоваться и клавишей <F7>.

Полный набор команд включает как команды самого редактора, так и команды, добавленные плагинами. Для выполнения команды следует набрать ее в окне командной строки и нажать клавишу <Enter>. В этом же окне отобразится и сообщение о результате выполнения команды. Если после ввода команды нажать клавишу <F7>, то окно ввода через несколько секунд будет скрыто. Для ввода новой команды и очистки окна следует еще раз нажать клавишу <F7>.

Инструмент командной строки имеет встроенную справочную систему. Для просмотра списка команд следует ввести команду

```
help list
```

Чтобы просмотреть справочную информацию по конкретной команде, нужно набрать

```
help команда
```

Здесь *команда* — название команды.

Командная строка предусматривает вызов уже выполненных команд — историю, для навигации по которой можно использовать клавиши <↑> и <↓>.

Работу с командной строкой редактора я продемонстрирую на примере. Пусть имеется открытый в редакторе `Kate` файл (рис. 6.27).

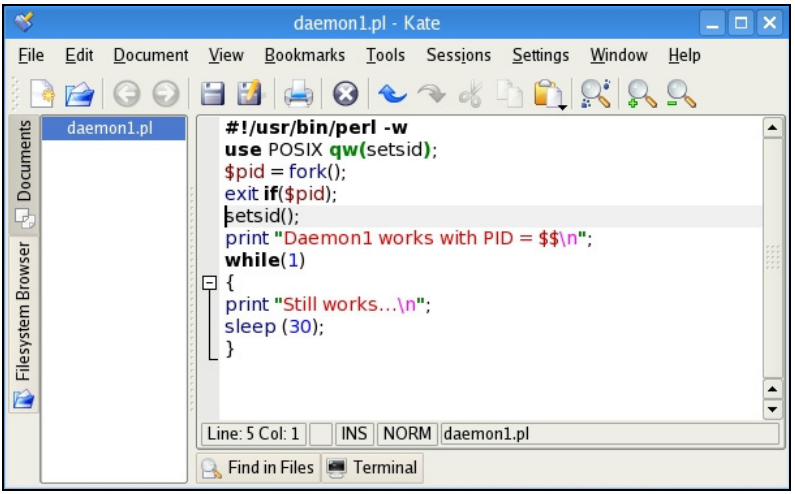


Рис. 6.27. Содержимое исходного файла перед выполнением команд

Вызовем командную строку (из меню или нажатием клавиши <F7>), после чего выполним последовательно три команды, представленные далее, причем выполнением команды `indent` выделим весь текст документа:

```
set-line-numbers on
```

```
indent
%s /print/PRINT/g
```

Первая команда устанавливает режим нумерации строк, вторая устанавливает отступы для всех строк документа, а третья выполняет поиск по всему файлу слова "print" и заменяет его на "PRINT". Вот как выглядит содержимое окна редактирования после выполнения этих трех команд (рис. 6.28).

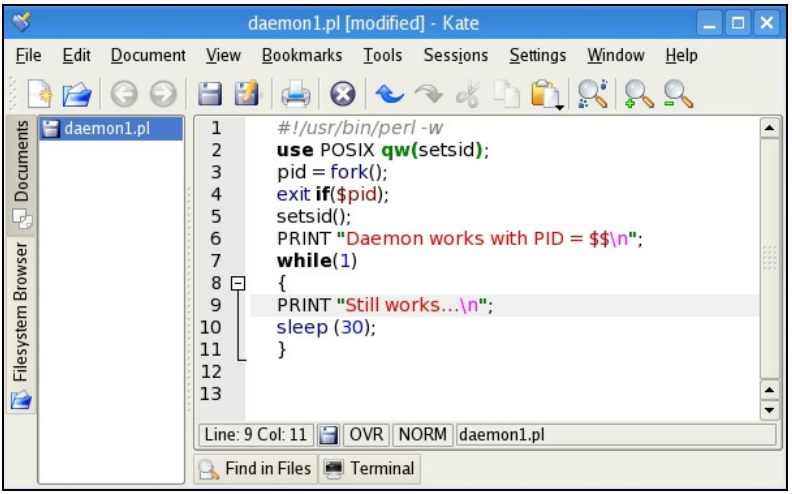


Рис. 6.28. Окончательный вид окна редактора

Кроме рассмотренных ранее возможностей, редактор `Kate` (так же как и `gedit`) может вызываться другими приложениями для обработки текстовых данных и формирования отчетов. Напомню, что `Kate` может функционировать в конвейере команд, а все языки высокого уровня, например, C и Perl, включают в себя функции для работы с программными каналами.

Рассмотрим пример использования редактора `Kate` в программе на языке Perl.

Предположим, что в системе имеются два файла с именами myfile и otherfile, содержимое которых показано далее:

```
# cat myfile
1-st String
2-nd String
3-rd String
First String
Second String
```

```
Third String
# cat otherfile
This is a first string
This is a 1-st string
This is a second string
This is a 2-nd string
This is a third string
This is a 3-rd string
```

Из этих файлов необходимо выделить все строки, которые не содержат символов цифр, и поместить их для дальнейшего редактирования в `Kate`.

Эту задачу можно решить, написав несложную программу на языке Perl (назовем ее kate_ipc.pl), исходный текст которой представлен далее:

```
#!/usr/bin/perl -w
die "Usage: $0 list_of_files\n" if (!@ARGV);
@myarray = ();
foreach $arg  (@ARGV)
{
  open(FD, "$arg") or die "open:$!";
  push(@myarray, "-------------------- $arg --------------------\n");
  @data = <FD>;
  push(@myarray, @data);
  close(FD);
}
open(KATE, "|kate --stdin") or die "open: $!";
foreach (@myarray)
 {
  next if ($_ =~ !/^.*[0-9].*$/);
  print KATE $_;
 }
close(KATE);
```

Здесь не будет рассматриваться алгоритм работы данной программы и смысл ее операторов — читателям не составит труда провести такой анализ самостоятельно, ознакомившись с основами языка Perl в *главе 12*. Обратите внимание на оператор

```
open(KATE, "|kate --stdin") or die "open: $!";
```

С его помощью реализуется механизм программного конвейера, в котором редактор `Kate` принимает входные данные от приложения.

Программа может принимать переменное число параметров. В данном случае для обработки файлов myfile и otherfile следует выполнить команду

```
# ./kate_ipc.pl myfile otherfile
```

Результат работы этой программы показан на рис. 6.29.

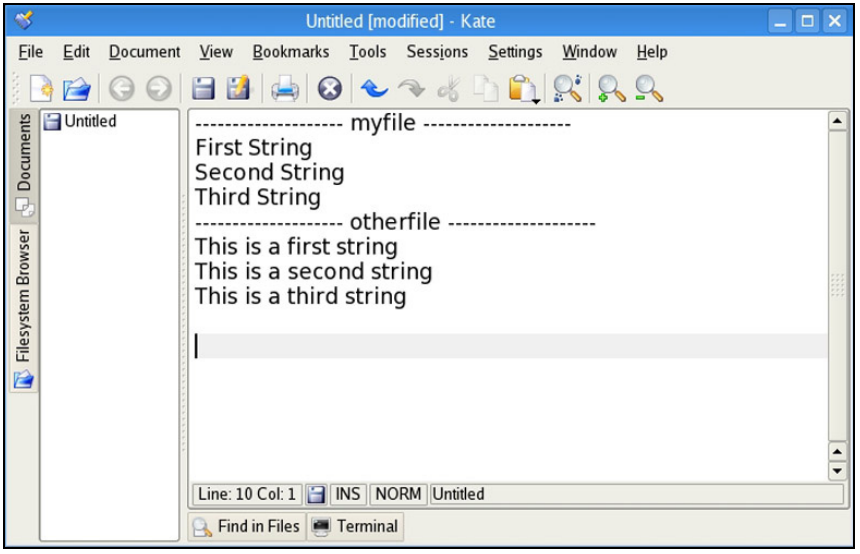


Рис. 6.29. Результат выполнения программы на Perl, использующей редактор `Kate`

Таковы в общих чертах особенности функционирования редактора `Kate`. Более подробная информация находится на сайте разработчика **http://www.kde.org/**, там же представлены и последние версии программного пакета.

Немного об инсталляции редактора `Kate`. Данная программа является частью проекта KDE (**http://www.kde.org/**) и входит в пакет kdebase, который находится на FTP-сайте **ftp://ftp.kde.org/pub/kde/**. Для установки редактора `Kate` следует выполнить следующие команды из каталога, где находится дистрибутив:

```
./configure
make
make install
```

Обычно инсталляция `Kate` выполняется без ошибок, но если таковые возникают, то можно обратиться на сайт разработчика за помощью, просмотрев соответствующие Web-страницы.

Глава 7

Процессы в UNIX

Понятие "процесс" является ключевым для понимания работы операционной системы UNIX. Процессом называют последовательность операций, выполняемых работающей программой и интерпретируемых центральным процессором как машинные инструкции, данные и стековые структуры. Например, при входе пользователя в операционную систему начинает выполняться процесс shell, при запуске какой-либо команды, например, `ls -l`, также порождается процесс. Примером создания нескольких процессов может служить случай, когда несколько пользователей запускают одновременно одну и ту же программу, в результате чего начинают работу несколько процессов. Обобщая, можно сказать, что всякий раз, когда вызывается команда UNIX или запускается пользовательская программа, порождается новый процесс. Из этого правила есть исключения, например, запуск команды `cd`, но подобные варианты рассматриваться не будут.

Любой процесс, порожденный основным процессом, называется дочерним или порожденным, а процесс, создавший порожденный процесс, называется родительским. При этом порожденный процесс наследует от родительского многие атрибуты, а его выполнение происходит независимо от родительского процесса. Каждый процесс имеет одного родителя и сам может порождать множество процессов.

В операционной системе UNIX одновременно выполняется множество процессов, обеспечивающих обслуживание многих пользователей и задач, что и лежит в основе многозадачного и многопользовательского режимов работы. Очень важной особенностью UNIX является то, что во время выполнения процесс может создавать или порождать новые процессы, что дает следующие преимущества:

- возможность создавать многозадачные приложения;

- возможность выполнения процесса в собственном адресном пространстве, что изолирует его от других процессов. При этом успешное или неуспешное выполнение процесса никак не влияет на родительский процесс;
- возможность для процесса создавать новые процессы, выполняющие отдельные программы, что значительно расширяет функциональные возможности операционной системы.

В общем случае структура данных и механизм выполнения процессов зависят от реализации операционной системы. Тем не менее, можно выделить некоторые общие черты, характеризующие функционирование процессов во всех операционных системах. Данные, доступные процессу, формируются ядром так, как показано на рис. 7.1.

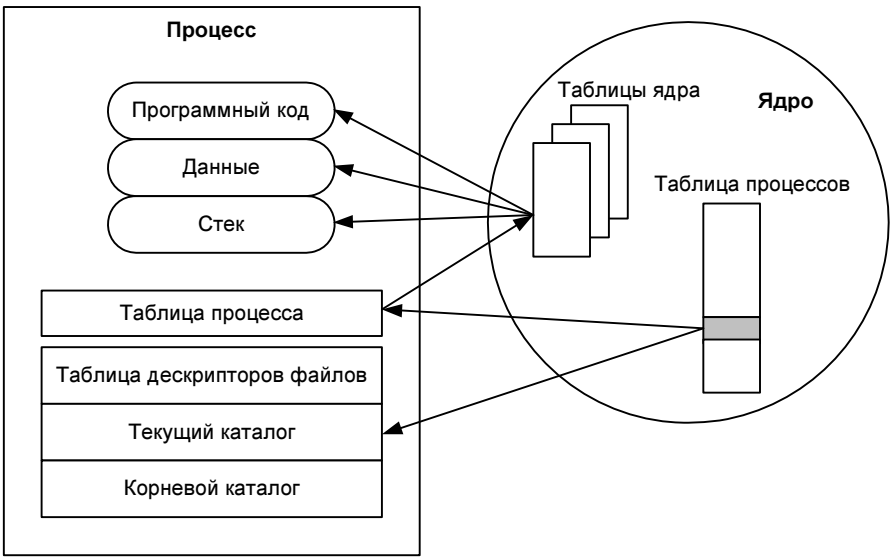


Рис. 7.1. Взаимодействие процесса и ядра UNIX

Операционная система манипулирует программным кодом (образом), а также разделами данных процесса, определяющими среду выполнения. Образ процесса размещается в сегменте кода и содержит реальные инструкции процессора, включающие как строки, скомпилированные и написанные пользователем, так и стандартный код, сгенерированный компилятором для системы. Этот код необходим для взаимодействия программы и операционной системы.

С процессом обычно связан терминал — логическое устройство, определяющее стандартные потоки для данного процесса: входной, выходной и поток сообщений об ошибках.

Данные, которыми манипулирует процесс, располагаются в областях памяти, выделяемых процессу динамически, т. е. по мере необходимости. Кроме того, каждому процессу выделяется стек. Он представляет собой область памяти, используемую для хранения локальных переменных программы и передачи параметров. При обращении процесса к функции или подпрограмме в стек отправляется новый фрейм (кадр). Каждый фрейм обязательно содержит указатель на базу предыдущего фрейма, что позволяет вернуть управление после вызова функции.

С практической точки зрения процесс в системе UNIX является объектом, создаваемым в результате выполнения системного вызова `fork()`. При этом процесс, выполняющий `fork()`, является родительским, а вновь созданный процесс — порожденным. Процесс с нулевым PID является особенным процессом, который создается при загрузке операционной системы.

Ядро системы идентифицирует каждый процесс по его номеру, который называется идентификатором процесса (Process ID, PID). Каждому процессу присваивается уникальный идентификатор PID, позволяющий ядру различать процессы. В момент создания нового процесса ядро присваивает ему следующий больший по числовому значению свободный идентификатор. Если достигнуто максимальное значение, равное 65 737, то очередной процесс получит минимальный свободный PID.

Завершение выполнения процесса инициируется системным вызовом `exit()`. Родительский процесс может проанализировать состояние завершения порожденного процесса, поскольку в системных таблицах сохраняется статистика выполнения процесса. Ядро операционной системы отслеживает окончание работы процесса, освобождая использовавшийся им идентификатор.

Вновь созданный процесс получает копии сегментов кода, данных и стека своего родительского процесса. Кроме того, порожденный процесс получает копию таблицы дескрипторов файлов, которая содержит ссылки на те же открытые файлы, что имеются у родительского процесса (рис. 7.2).

Это позволяет как родительскому, так и порожденному процессу использовать один и тот же дескриптор при обращении к открытому файлу. Например, если родительским процессом открыты файлы, и им присвоены дескрипторы `fdesc1`, `fdesc2` и `fdesc3` (см. рис. 7.2), то порожденный процесс наследует копии этих дескрипторов и может выполнять операции с соответствующими файлами. Кроме того, закрытие дескриптора файла в родительском процессе не приведет к автоматическому закрытию копии дескриптора в порожденном процессе, т. е. порожденный процесс сможет и дальше выполнять операции с файлом. Верно и обратное — если дескриптор файла закрывается в порожденном процессе, то это никак не сказывается на файловых операциях в родительском процессе.

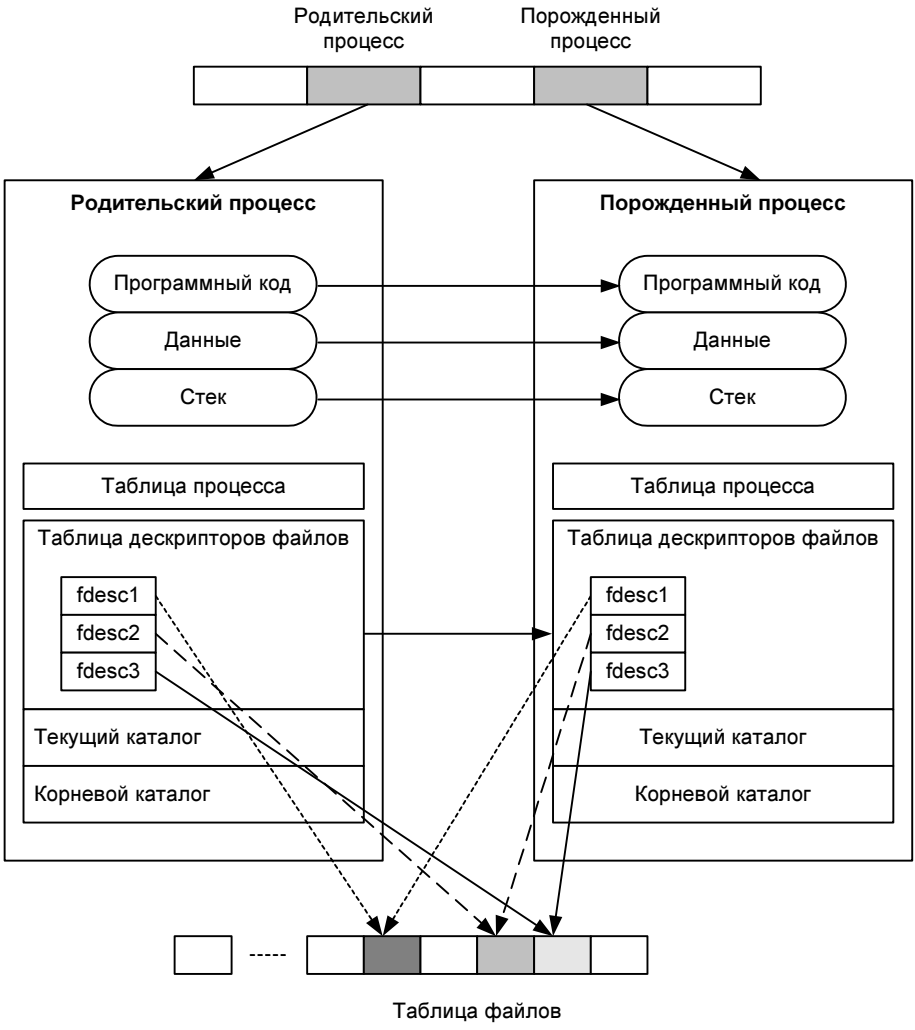


Рис. 7.2. Структуры данных родительского и порожденного процессов

Кроме вышеперечисленных, ядро присваивает процессу и другие атрибуты, некоторые из них приведены далее:

- реальный идентификатор владельца (rUID), соответствующий идентификатору пользователя, который создал родительский процесс. Реальный идентификатор нужен для того, чтобы ядро отслеживало, кто и какие процессы создает в системе;

- реальный идентификатор группы (rGID), соответствующий идентификатору группы, к которой принадлежит пользователь, создавший родительский процесс. Этот атрибут используется ядром системы для того, чтобы отслеживать, какая группа порождает те или иные процессы;
- эффективный идентификатор пользователя (eUID), обычно совпадающий с rUID, за исключением случая, когда у файла, который был выполнен для создания процесса, установлен флаг set-UID (обычно это делается при помощи системного вызова `chmod()` или команды `chmod` командного интерпретатора). Если флаг set-UID установлен, то эффективный идентификатор eUID становится равным UID файла. Это позволяет процессу обращаться к файлам и создавать новые файлы, пользуясь теми же привилегиями, что и у владельца файла;
- эффективный идентификатор группы владельца (eGID), обычно совпадающий с rGID, за исключением случая, когда у файла, который был выполнен для создания процесса, установлен флаг set-GID. В этом случае eGID процесса становится равным GID файла, что позволяет процессу обращаться к файлам и создавать файлы, пользуясь привилегиями группы, к которой относится программный файл;
- текущий каталог, представленный номером индексного дескриптора, указывающего на рабочий каталог;
- корневой каталог, представленный номером индексного дескриптора, указывающего на корневой каталог;
- параметры обработки сигналов;
- управляющий терминал процесса;
- значение приоритета процесса.

Родительский и порожденный процессы отличаются рядом параметров, например:

- идентификаторами процессов (PID), представляющих собой целочисленные значения, уникальные для операционной системы;
- идентификаторами родительских процессов (PPID).

В процессе загрузки системы ядро UNIX загружается в память и немедленно запускает необходимые процессы, наиболее важным из которых является процесс `init`, выполняющий инициализацию операционной системы. Этот процесс имеет идентификатор, равный 1, является предком любого другого процесса в системе и связан с каждым процессом особым образом.

Следует сказать, что создание процесса в операционной системе UNIX не всегда заканчивается успехом. Чаще всего это связано с некорректным за-

вершением системного вызова `fork()`. При этом невозможность создания процесса может быть вызвана следующими причинами:

- системным ограничением на общее количество выполняемых процессов;
- системным ограничением на количество одновременно созданных процессов для одного пользователя;
- недостаточным объемом системной памяти.

При успешном завершении порожденному процессу возвращается значение 0, а родительскому процессу — идентификатор порожденного процесса. В случае ошибки родительскому процессу возвращается –1, новый процесс не создается, и переменной `errno` присваивается код ошибки.

Во время выполнения или во время ожидания процессы находятся в виртуальной памяти, которая отображается на конкретные адреса физической памяти. Следует учитывать, что процессам доступна только часть физической памяти, поскольку определенный объем физической памяти резервируется для ядра операционной системы. Пользователи могут получить доступ только к виртуальной памяти процессов. При необходимости страницы памяти процессов откачиваются из физической памяти на диск, в область подкачки (свопинга). При обращении к странице в виртуальной памяти, если она не находится в физической памяти, происходит ее подкачка с диска.

Все процессы, выполняющиеся в операционной системе UNIX, можно разделить на системные, процессы-демоны и прикладные процессы.

Системные процессы являются частью ядра и всегда расположены в оперативной памяти. Они не имеют выполняемых файлов и запускаются особым образом при инициализации ядра системы. Исполняемые инструкции и данные таких процессов постоянно находятся в ядре системы, поэтому они могут вызывать другие функции и обращаться к данным, недоступным для остальных процессов. К системным процессам относится и процесс начальной инициализации `init`, являющийся прародителем всех остальных процессов. Несмотря на то, что `init` не является частью ядра и запускается из выполняемого файла, его функционирование критически важно для всей системы в целом. Остановимся на этом процессе более подробно.

Программа /sbin/init, запускающая процесс `init`, порождает процессы для запуска системы на основе записей, находящихся в файле /etc/inittab. При этом процесс `init` анализирует записи в файле /etc/inittab и определяет последовательность запуска, остановки и перезапуска остальных процессов.

Формат каждой записи файла /etc/inittab следующий:

```
id:runlevel:action:process
```

Здесь:

- `id` — уникальный идентификатор из одного или двух символов;
- `runlevel` — уровень запуска, на котором должна обрабатываться эта запись. Уровни запуска соответствуют конфигурации процессов в системе и представлены числами от 0 до 6. Например, если система работает на уровне запуска 1, обрабатываются только записи, в поле `runlevel` которых указано значение 1;
- `action` — способ запуска процесса:
  - `respawn` — если процесс не существует, то необходимо его создать, а когда процесс завершит существование, то перезапустить его. Если процесс уже выполняется, продолжить просмотр файла /etc/inittab;
  - `wait` — процесс `init` ожидает завершения процесса, запущенного на этом уровне. Для всех последующих просмотров файла /etc/inittab процессом `init` на том же уровне данная запись игнорируется;
  - `once` — процесс запускается, причем `init` не ожидает его завершения. При завершении выполнения процесс не перезапускается;
  - `boot` — процесс выполняется только при первом просмотре файла /etc/inittab в момент загрузки процесса `init`. Процесс `init`, запустив процесс, не будет ожидать его завершения; по завершению процесса `init` его не перезапускает;
  - `bootwait` — процесс начинает выполняться только на первом переходе процесса `init` с однопользовательского в многопользовательский режим, после того как система загружена. Процесс `init` запускает процесс, ожидает его завершения, после чего процесс не перезапускается;
  - `powerfail` — процесс выполняется только в том случае, если процесс `init` получает сигнал сбоя по питанию;
  - `off` — выполняющемуся процессу посылается сигнал `SIGTERM`, после чего процесс завершается принудительно посылкой сигнала `SIGKILL`. Если процесс не выполняется, то запись игнорируется;
  - `ondemand` — процесс выполняется так же, как и для опции `respawn`;
  - `initdefault` — выполняется только при первом запуске `init`. Процесс `init` использует эту запись, чтобы определить, на какой уровень выполнения первоначально переходить;
  - `sysinit` — процесс выполняется до того, как `init` выполнит обращение к консоли. Процесс `init` ждет завершения процесса, прежде чем продолжить работу;

- process — имя команды, подлежащей выполнению. Задает команду, которую надо выполнить. Перед этим полем необходимо указывать команду exec. В поле process можно указывать любую допустимую команду.

Записи разделяются символами новой строки; но если перед символом новой строки присутствует обратный слэш (\), то запись продолжается на следующей строке. Максимальная длина записи составляет 512 символов, при этом в поле process можно использовать комментарии. На количество записей в файле inittab никаких ограничений не налагается (только на размер одной записи).

При получении процессом init запроса на изменение уровня выполнения всем процессам, для которых в поле runlevel не указан целевой уровень, посылается сигнал SIGTERM, а через 5 секунд их работа принудительно прерывается сигналом SIGKILL. Как уже отмечалось, поле runlevel может задавать несколько уровней выполнения для процесса в виде любой комбинации значений от 0 до 6. Если уровень выполнения не задан, предполагается, что процесс может работать на всех уровнях выполнения.

Новые записи можно добавлять в файл /etc/inittab в любой момент. Однако процесс init будет ждать наступления одного из перечисленных выше условий, прежде чем просматривать файл /etc/inittab. Для немедленной инициализации только что добавленного процесса нужно выполнить одну из команд

```
init Q
init q
```

Эти команды вынуждают процесс init немедленно перечитать файл /etc/inittab.

При получении процессом init сигнала о сбое питания он ищет в файле /etc/inittab специальные записи типа powerfail и powerwait. Соответствующие этим записям процессы запускаются (если уровень запуска runlevel позволяет это сделать) перед выполнением любой последующей команды. Процесс init выполняет необходимые действия по завершению работы операционной системы и сохранению информации при отключении питания.

Вот пример файла /etc/inittab, используемого операционной системой Solaris 10:

```
# Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# The /etc/inittab file controls the configuration of init(1M); for more
# information refer to init(1M) and inittab(4).  It is no longer
# necessary to edit inittab(4) directly; administrators should use the
```

```
# Solaris Service Management Facility (SMF) to define services instead.
# Refer to smf(5) and the System Administration Guide for more
# information on SMF.
#
# For modifying parameters passed to ttymon, use svccfg(1m) to modify
# the SMF repository. For example:
#
#       # svccfg
#       svc:> select system/console-login
#       svc:/system/console-login> setprop ttymon/terminal_type = "xterm"
#       svc:/system/console-login> exit
#
#ident  "@(#)inittab    1.41    04/12/14 SMI"
ap::sysinit:/sbin/autopush -f /etc/iu.ap
sp::sysinit:/sbin/soconfig -f /etc/sock2path
smf::sysinit:/lib/svc/bin/svc.startd >/dev/msglog 2<>/dev/msglog </dev/console
p3:s1234:powerfail:/usr/sbin/shutdown -y -i5 -g0 >/dev/msglog 2<>/dev/msglog
```

Демоны (daemons) — это неинтерактивные процессы, которые запускаются обычным образом как выполняемые файлы и выполняются, как правило, в фоновом режиме. Процессы-демоны играют чрезвычайно важную роль в функционировании операционной системы и, как правило, запускаются после инициализации ядра, обеспечивая работу различных подсистем UNIX: систем терминального доступа и печати, сетевых сервисов и т. д. Мы рассмотрим более подробно функционирование демонов далее в этой главе.

К прикладным процессам относится большинство процессов, выполняющихся в системе. Как правило, это процессы, созданные в контексте пользовательского сеанса работы, важнейшим из которых является командный интерпретатор shell, обеспечивающий выполнение команд пользователя в системе UNIX. Пользовательские процессы могут выполняться как в интерактивном (приоритетном), так и в фоновом режимах. Интерактивные процессы монопольно владеют терминалом, и пока такой процесс не завершит свое выполнение, пользователь не имеет доступа к командной строке.

7.1. Взаимодействие процессов

Проанализируем практические аспекты создания и взаимодействия процессов и начнем с выполнения файловых операций в родительском и порожденном процессах.

Предположим, что в родительском процессе с помощью системного вызова `open()` открывается пустой файл с именем TEXT для чтения/записи, и ему присваивается дескриптор `fdesc1`:

```
fdesc = open("TEXT", O_RDWR);
```

В порожденном процессе этот же файл будет доступен для чтения/записи с тем же дескриптором, что и в родительском процессе. Кроме того, указатель текущей позиции в файле в порожденном процессе будет таким же, что и в родительском в момент создания нового процесса. Пусть родительский процесс перед созданием нового процесса с помощью системного вызова `fork()` записывает в файловый дескриптор строку `STRING FROM PARENT PROCESS`:

```
. . .
char *str1 = "STRING FROM PARENT PROCESS";
write(fdesc1, str1, strlen(str1);
```

Создадим новый процесс, в котором выполняется запись в файл TEXT строки `STRING FROM CHILD PROCESS` при помощи операторов

```
. . .
char *str2 = "STRING FROM CHILD PROCESS";
fork();
. . .
write(fdesc1, str2, strlen(str2);
. . .
```

Здесь при выполнении операции записи используется один и тот же дескриптор `fdesc1` файла TEXT. По завершению операций записи в дескриптор `fdesc1` результирующая строка может иметь одно из двух представлений:

```
STRING FROM PARENT PROCESSSTRING FROM CHILD PROCESS
STRING FROM CHILD PROCESSSTRING FROM PARENT PROCESS
```

В общем случае предсказать, какая из этих двух строк будет записана в файл, нельзя. Это объясняется тем, что в момент создания нового процесса он начинает выполняться вне контекста родительского процесса и синхронизируется только ядром операционной системы. При этом в зависимости от реализации UNIX порожденный процесс может выполниться как раньше, так и позже родительского.

Если теперь в порожденном процессе закрыть файловый дескриптор `fdesc1` с помощью системного вызова `close(fdesc1)`, то это не воспрепятствует выполнению файловых операций с этим дескриптором в родительском процессе, поскольку был закрыт дубликат `fdesc1`!

Это легко проверить, выполнив запись в файловый дескриптор какого-нибудь текста:

```
. . .
char *str3 = "Writing completed.";
write(fdesc1, str3, strlen(str3);
```

После закрытия последнего дескриптора файл TEXT можно прочитать командой `cat` и убедиться, что там содержится строка:

```
#cat TEXT
STRING FROM PARENT PROCESS STRING FROM CHILD PROCESSWriting completed.
```

Еще один важный момент, на который следует обратить внимание, — то, как обрабатываются данные в родительском и порожденном процессах. Я уже упоминал о том, что порожденный процесс получает копию области программ, данных и стека, которая располагается в отдельной от родительского процесса области виртуальной памяти, выделяемой порожденному процессу. Это означает, что в общем случае порожденный процесс не может изменить данные родительского процесса, и, наоборот, родительский процесс не имеет доступа к данным порожденного им процесса. В практическом плане это значит, что оба процесса оперируют локальными, т. е. независимыми данными. Этот тезис проиллюстрирован на рис. 7.3.

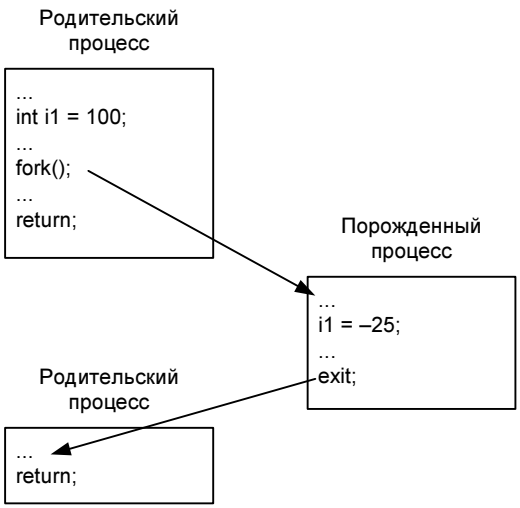


Рис. 7.3. Манипуляции данными в родительском и порожденном процессах

Как видно из рисунка, целочисленная переменная `i1` в родительском процессе принимает значение 100. Затем при помощи системного вызова `fork()`

создается новый процесс, в котором переменная `i1` получает значение –25. Если просмотреть значение `i1` в родительском процессе после завершения порожденного процесса, то оно останется равным 100. Это означает, что порожденный процесс, вообще говоря, манипулирует другой переменной, т. е. другой ячейкой памяти. При этом в момент создания нового процесса данная ячейка памяти содержала значение 100, переданное родительским процессом.

Возникает закономерный вопрос, каким образом сделать данные, обрабатываемые одним процессом, доступными другому процессу? Для этих целей в операционной системе UNIX предусмотрен механизм обмена, известный как неименованный канал. Неименованный канал (часто его называют анонимным каналом) создается при помощи системного вызова `pipe()` и обеспечивает коммуникации между родительским и порожденным процессами, а также между порожденными процессами одного и того же родительского процесса.

Функция `pipe()` создает файл канала, который является временным буфером и используется для того, чтобы вызывающий процесс мог записывать и считывать данные другого процесса. Файлу канала имя не присваивается, отсюда и название — неименованный канал. Канал освобождается после того, как все процессы закрывают файловые дескрипторы, которые ссылаются на данный канал. Взаимодействие родительского и порожденного процессов посредством неименованного канала показано на рис. 7.4.

Процесс взаимодействия родительского и порожденного процессов можно описать следующим образом:

1. Вначале родительский процесс выполняет системный вызов `pipe()` для создания канала (1).
2. С помощью системного вызова `fork()` создается новый процесс, который получает копии дескриптора канала от родительского процесса (2).
3. После этого порожденный процесс может читать данные из канала или записывать их в канал (3). В данном случае выполняется запись данных в канал с использованием системного вызова `write()`.
4. Родительский процесс также может принимать данные из канала от порожденного процесса или отправлять ему данные через этот же канал (4). В данном случае родительский процесс читает данные из канала посредством системного вызова `read()`.

Оптимальный вариант использования неименованных каналов — когда в межпроцессных коммуникациях участвуют два процесса, при этом один процесс является отправителем данных, а другой — получателем. Следует сказать, что системный вызов `pipe()` используется командным интерпретатором shell для реализации программного канала (обозначается как `|`) с целью соединения стандартного вывода одного процесса со стандартным вводом другого.

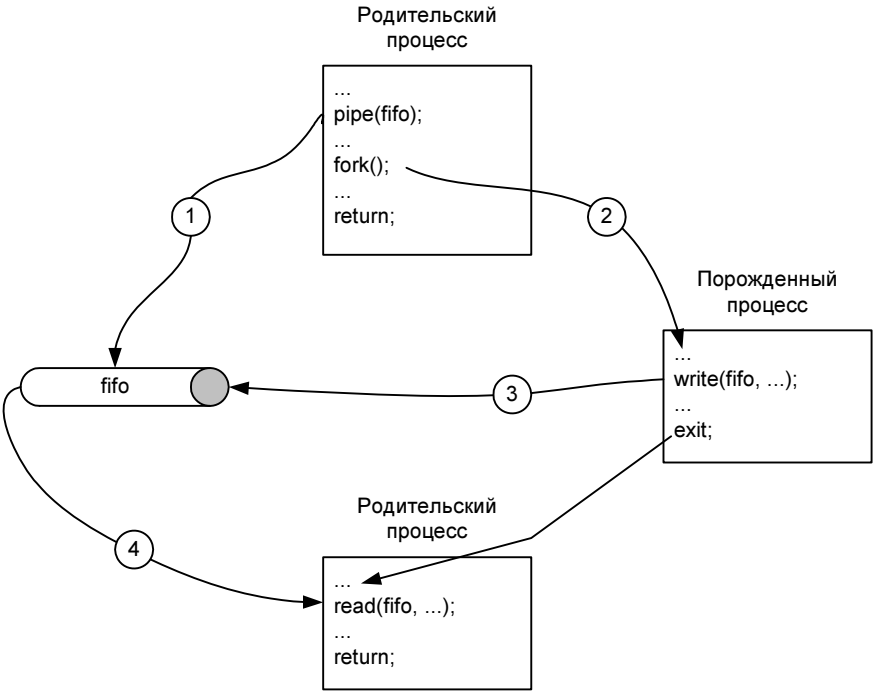


Рис. 7.4. Взаимодействие процессов посредством неименованных каналов

До сих пор мы рассматривали взаимодействие процессов, один из которых является родительским, а остальные — порожденными. В операционной системе UNIX большинство процессов запускается и работает автономно от других процессов. Взаимодействие между такими процессами осуществляется по иным принципам, чем те, которые мы только что рассматривали.

Одним из распространенных способов межпроцессного взаимодействия между обособленными процессами является обмен данными через именованные каналы (named pipes). Очень часто именованные каналы называют именованными сокетами или сокетами UNIX, поскольку они используют принципы обмена данными по сети по протоколу TCP или UDP. Именованные каналы позволяют нескольким процессам передавать данные через одно и то же соединение. Имена сокетов UNIX похожи на имена файлов в файловой системе, в то время как сами сокеты UNIX представляют собой специальные файлы, создаваемые на диске. Если для взаимодействия процессов используется потоковая модель (SOCK_STREAM), то один из процессов ожидает запроса приема или передачи данных от другого процесса, после чего устанавливается соединение и происходит обмен данными. В этом случае

ожидающий соединения процесс называется сервером, а процесс, запрашивающий данные, — клиентом. Это весьма условное деление, поскольку один и тот же процесс может быть как сервером, так и клиентом. Тем не менее, такая упрощенная модель взаимодействия позволяет понять базовые принципы использования потоковых сокетов UNIX.

При использовании дейтаграммных сокетов UNIX (SOCK_DGRAM) процессы взаимодействуют между собой без установления предварительного соединения, при этом нельзя гарантировать, что все данные будут переданы или получены, в отличие от модели SOCK_STREAM. Процессы, получающие и/или передающие данные друг другу, должны сами заботиться о целостности данных. Преимуществом использования дейтаграммных сокетов по сравнению с потоковыми является более быстрая передача/прием данных, поскольку не тратится время на установление соединения.

Упрощенно модель взаимодействия процессов при помощи сокетов UNIX можно представить так, как показано на рис. 7.5 (здесь используются потоковые сокеты).
Проанализируем более подробно механизм взаимодействия процессов, представленный на рисунке.

Процесс-сервер создает сокет UNIX с именем `mysocket`, присваивая ему дескриптор `fdsock`. Для этого используется системный вызов `socket()`, после чего все дальнейшие операции с сокетом выполняются посредством обращения к его дескриптору. Системный вызов `bind()` связывает с сокетом область данных, в которую помещаются параметры сокета (тип сокета, механизм обмена данными, имя файла сокета). После этого сокет устанавливается в режим прослушивания соединений при помощи системного вызова `listen()`, ожидая подключения клиентов (1).

При получении требования об установлении соединения от процесса-клиента сервер создает новое соединение с помощью системного вызова `accept()` и выполняет обработку данных и иные действия, после чего обычно закрывает соединение (2).

Процессы-клиенты реализованы проще, чем сервер: для них требуется создать сокет с помощью системного вызова `socket()`, указав при этом путь к файлу (в данном случае `mysocket`). Затем следует просто выполнить подключение к сокету, используя системный вызов `connect()`. Далее выполняется прием/передача, а также обработка данных (3, 4).

Хочу сделать важное замечание: процесс-клиент не может создавать физический файл сокета, это должен сделать сервер до того, как начнут поступать клиентские запросы! Если этого не сделать, то процессы-клиенты будут возвращать ошибки подключения.

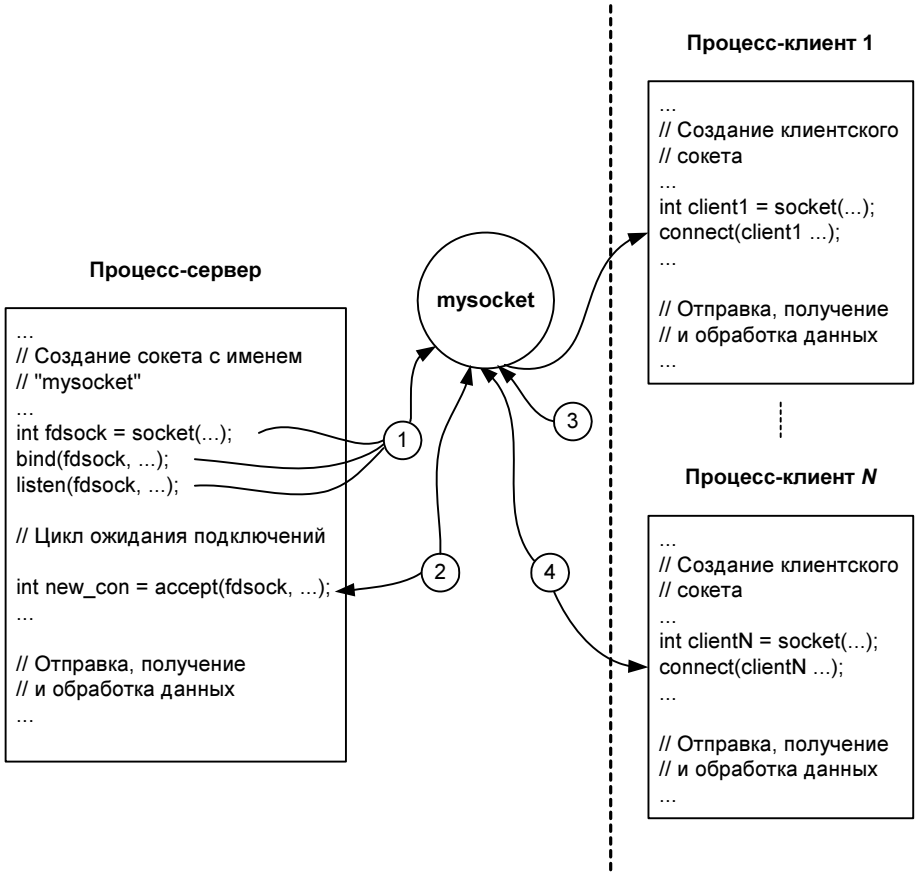


Рис. 7.5. Взаимодействие процессов через потоковые сокеты

Межпроцессное взаимодействие с использованием дейтаграммных сокетов (SOCK_DGRAM) выполняется намного проще, чем взаимодействие при помощи потоков. В этом случае серверу не нужно прослушивать сокет в ожидании входящих соединений, что упрощает алгоритм обработки данных, правда, с одновременным уменьшением надежности обмена данными.

Для межпроцессных коммуникаций можно использовать и механизмы сетевого обмена данными с помощью протокола TCP/IP или UDP, причем даже на локальной системе. Взаимодействие посредством сокетов UNIX очень широко используется в межпроцессных коммуникациях. Процесс-сервер, принимающий запросы от клиентов, может работать в одном из двух режимов: блокирующем или неблокирующем. В блокирующем режиме в любой момент времени обрабатывается только один клиентский запрос, причем ос-

тальные клиенты должны ждать завершения операции и закрытия соединения. Для неблокирующего режима допускается обработка нескольких клиентов одновременно.

В операционной системе UNIX существуют и другие механизмы межпроцессного взаимодействия, например, обмен данными с использованием общих областей памяти или механизм удаленных вызовов процедур (RPC), которые ввиду сложности и специфики применения рассматриваться здесь не будут.

7.2. Демоны UNIX

Процессы-демоны не связаны с терминалом и бóльшую часть времени ожидают, пока тот или иной процесс запросит определенную услугу.

Процесс-демон можно охарактеризовать следующим образом:

- он выполняется в фоновом режиме;
- он не связан ни с одним терминалом;
- функционирование процесса-демона не зависит от того, какой пользовательский сеанс работает в данный момент;
- он не получает сигналы терминала (например, от нажатия комбинации клавиш <Ctrl>+<C>).

В операционных системах UNIX обычно выполняется много процессов-демонов. Большинство серверных приложений выполняется как демоны, например:

- Web-сервер (`httpd`);
- почтовый сервер (`sendmail`);
- суперсервер (`inetd`);
- сервер регистрации событий в системе (`syslogd`);
- сервер печати (`lpd`);
- процессы `routed` и/или `gated`, выполняющие функции маршрутизации.

Поскольку процессы-демоны не связаны ни с одним терминалом, то, как правило, они используют файловый вывод или формируют записи для журнала регистрации событий.

Любая версия операционной системы UNIX включает так называемый суперсервер, который в большинстве систем имеет имя `inetd`. Суперсервер `inetd` выполняет специальные задачи, критически важные для функционирования UNIX:

- запускает код инициализации для групп серверов;

- ожидает поступления входящих запросов, направленных группам серверов;
- при поступлении запросов сервер `inetd` запускает соответствующий сервер и передает ему запрос.

Для запуска каждого сервера демон `inetd` должен располагать следующей информацией:

- номер порта и транспортный протокол;
- какие флаги установлены: `wait`/`nowait`;
- регистрационное имя, под которым процесс должен запускаться;
- путевое имя программного файла, запускающего сервер;
- аргументы командной строки программы-сервера.

Информацию о запускаемых программах-серверах суперсервер `inetd` считывает из файла /etc/inetd.conf, пример записи из которого представлен далее:

```
# comments start with #
echo    stream   tcp nowait root  internal
echo    dgram    udp wait   root  internal
chargen stream   tcp nowait root  internal
chargen dgram    udp wait   root  internal
ftp     stream   tcp nowait root  /usr/sbin/ftpd ftpd -l
telnet  stream   tcp nowait root  /usr/sbin/telnetd telnetd
finger  stream   tcp nowait root  /usr/sbin/fingerd fingerd
# Authentication
auth    stream   tcp nowait nobody /usr/sbin/in.identd in.identd -l -e -o
# TFTP
tftp    dgram    udp wait root /usr/sbin/tftpd tftpd -s /tftpboot
```

Если для запускаемого процесса указан флаг `wait`, то это означает, что `inetd` не принимает запросы от новых клиентов, пока порожденный им процесс-сервер не завершится. Обычно для TCP-серверов указывается флаг `nowait`, что позволяет суперсерверу `inetd` запускать несколько копий TCP-сервера. Что же касается UDP-серверов, то большинство из них запускается с флагом `wait`.

Следует заметить, что серверы, к которым выполняется много обращений, обычно не запускаются через `inetd`. К таким серверам можно отнести Web-серверы, почтовые и некоторые другие сетевые серверы. Кроме того, многие серверы позволяют указывать при их запуске через `inetd` опции командной строки.

В некоторых версиях UNIX вместо классического суперсервера `inetd` используется сервер `xinetd`, который отличается от `inetd` только схемой конфигурирования.

Поскольку принцип работы процессов-демонов отличается от функционирования обычных процессов, то вначале рассмотрим некоторые теоретические аспекты разработки демонов.

Обычный процесс во время выполнения получает сигналы от терминала, с которым он связан и который наследуется от родительского процесса. Процесс-сервер, а таковым является демон, не получает сигналы от родительского процесса, поэтому процесс-демон должен быть каким-то образом отсоединен от управляющего терминала. Кроме того, в операционных системах UNIX процессы выполняются в контексте группы процессов, а при создании нового процесса наследуется группа, в которой выполняется родительский процесс. Для работы в режиме демона порожденный процесс должен находиться в новой группе и выполняться в отдельной сессии.

Для выполнения этих условий работающий процесс должен осуществить системный вызов `setsid()`, который создает новую сессию и новую группу процессов, если вызывающий данную функцию процесс не является лидером группы процессов. После выполнения системного вызова `setsid()` вызывающий процесс становится лидером новой сессии и лидером группы процессов, и, кроме того, такой процесс не имеет управляющего терминала. Идентификатор группы процессов для вызывающего процесса становится равным идентификатору самого вызывающего процесса. Таким образом, вызывающий процесс становится единственным в новой группе процессов и единственным процессом в новой сессии.

Функция `setsid()` имеет следующий синтаксис:

```
pid_t setsid(void)
```

Здесь `pid_t` — значение идентификатора группы процессов, которое возвращается вызывающему процессу.

В практическом плане для создания процесса-демона необходимо выполнить два системных вызова — `fork()` и `setsid()`. Системный вызов `fork()` создает порожденный процесс, после чего следует вызвать функцию `setsid()`. "Осиротевший" процесс-потомок становится потомком процесса `init` и полностью отсоединяется от родительского процесса, продолжая выполнение в фоновом режиме. Для процесса-демона желательно установить и определенные параметры среды выполнения (корневой каталог, маску файлов). В простейшем случае процесс-демон создается несколькими строками программного кода:

```
. . .
pid_t pid;
```

```
pid = fork();       // создается процесс-потомок
if (pid) exit (1); // завершить родительский процесс
setsid();       // сделать процесс-потомок лидером группы и лидером сеанса
. . .
```

Рассмотрим пример программы-демона (назовем ее `simpled2`), исходный текст которой приведен далее. Демон записывает каждые 30 секунд строку с информацией о количестве активизаций в файл simpled2.LOG:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int main(void)
{
  int pid = fork();
  if (pid) exit(0);
  setsid();

  char buf[128];
  int bytesRead;
  int fd, cnt = 0;

  pid_t pid_child = getpid();
  bytesRead = sprintf(buf, "%d", (int)pid_child);

  int fpid = open("simpled2.PID", O_RDWR | O_CREAT, 0644);
  write(fpid, buf, bytesRead);
  close(fpid);
  printf("Daemon simpled2 launched with PID=%d\n", (int)pid_child);
  close(0);
  close(1);
  close(2);
  while (true)
  {
   fd = open("simpled2.LOG",  O_RDWR | O_CREAT | O_APPEND, 0644);
   bytesRead = sprintf(buf, "%s %d %s\n", "Daemon burst ", ++cnt, " times");
   write(fd, buf, bytesRead);
```

```
   close(fd);
   sleep(30);
  }
return 0;
}
```

Проанализируем исходный текст программы. С помощью следующих операторов создается процесс-демон:

```
int pid = fork();
if (pid) exit(0);
setsid();
```

Для того чтобы можно было уничтожить процесс-демон без определения PID, его идентификатор сохраняется с помощью системного вызова `getpid()` в переменной `pid_child`. Затем `pid_child` преобразуется к строковому представлению и записывается в файл simpled2.PID. Все эти операции выполняет следующий фрагмент кода:

```
pid_t pid_child = getpid();
bytesRead = sprintf(buf, "%d", (int)pid_child);

int fpid = open("simpled2.PID", O_RDWR | O_CREAT, 0644);
write(fpid, buf, bytesRead);
close(fpid);
```

После этого в бесконечном цикле `while(1)` каждые 30 секунд выполняется запись строки, предварительно сформированной в буфере `buf`, в файл simpled2.LOG.

При запуске программа отображает на экране дисплея идентификатор процесса:

```
bash-3.00# ./simpled2
Daemon simpled2 launched with PID=713
```

Легко проверить, что демон `simpled2` выполняется:

```
bash-3.00# ps -ef|grep TTY;ps -ef|grep simpled2
UID   PID  PPID   C    STIME TTY         TIME CMD
root   715   710   0 16:47:59 pts/4       0:00 grep TTY
root   713     1   0 16:47:26 ?           0:00 ./simpled2
root   717   710   0 16:47:59 pts/4       0:00 grep simpled2
```

Из результата выполнения команд `ps` видно, что родительским процессом (поле PPID) для `simpled2` является процесс `init`. Обратите внимание на то, что демон не имеет управляющего терминала (поле `TTY`). Кроме того, демон

simpled2 имеет PID, равный 713. Это же значение записано и в файл simpled2.PID, созданный при запуске программы:

```
bash-3.00# cat simpled2.PID
713bash-3.00#
```

Кроме файла simpled2.PID, в рабочем каталоге программы создается файл simpled2.LOG, в который каждые 30 секунд добавляются записи. Если проверить содержимое этого файла через некоторое время после запуска демона simpled2, то оно может выглядеть примерно так:

```
bash-3.00# cat simpled2.LOG
Daemon burst  1  times
Daemon burst  2  times
Daemon burst  3  times
Daemon burst  4  times
Daemon burst  5  times
Daemon burst  6  times
Daemon burst  7  times
Daemon burst  8  times
```

Уничтожить процесс-демон можно командой kill, послав ему сигнал с идентификатором 9 (SIGKILL):

```
bash-3.00# kill -9 713
```

В команде kill вместо явного указания PID выполняющегося процесса можно использовать значение, записанное в файл simpled2.PID:

```
bash-3.00# kill -9 `cat simpled2.PID`
```

На этом рассмотрение принципов функционирования программ-демонов мы закончим и перейдем к анализу практических аспектов взаимодействия пользовательских программ и операционной системы UNIX.

7.3. Программный интерфейс пользователя

В этом разделе приводятся примеры программного кода на языке C, демонстрирующие практическую сторону взаимодействия процессов. Исходные тексты программ максимально упрощены, чтобы не обременять читателя подробностями (анализ кодов завершения, усложненный алгоритм и т. д.). Для проверки работоспособности примеров использовались операционные системы Solaris 5.11 и Red Hat Linux 9.

Во всех примерах, как и в предыдущих главах, используются системные вызовы (функции API) стандарта POSIX, поэтому приведенные примеры будут работать практически без изменений во всех версиях UNIX. Единственно, на что следует обратить внимание — это файлы заголовков (расширение h), в которых описаны те или иные функции. Тем не менее, подавляющее большинство примеров будет работать вообще без каких-либо изменений в исходных текстах.

Пример 1. Здесь приводится исходный текст программы, демонстрирующей взаимодействие основного (родительского) и созданного им нового (порожденного) процесса:

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
 int pid;
 if ((pid = fork()) == 0)
 {
  printf("Child process works!\n");
  return 0;
 }
 else if (pid > 0)
  printf("Parent process works!\n");
 else
  printf("Cannot create child process!\n");
 return 0;
}
```

Здесь при помощи системного вызова `fork()` порождается новый процесс:

```
if ((pid = fork()) == 0)
```

Все, что делает порожденный процесс — выводит на консоль сообщение "Child process works!", после чего завершается с помощью оператора `return 0`. Это очень важный момент — все порожденные процессы, кроме таких специальных случаев, как, например, процессы-демоны, должны завершаться оператором `return` или системным вызовом `exit()`. В зависимости от реализации операционной системы сообщение "Child process works!" может появиться как раньше, так и позже сообщения "Parent process works!"

Пример 2. Это модифицированный вариант примера 1. Здесь в исходный текст программы (назовем ее `process2`) добавлен системный вызов `wait()`,

позволяющий приостановить родительский процесс до завершения порожденного процесса. Для большей наглядности в порожденном процессе используется функция `sleep()`, позволяя задержать выполнение программы на 20 секунд:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
  int pid, status;
  if ((pid = fork()) == 0)
  {
    printf("Child process works!\n");
    sleep(20);
    return 0;
  }
  else if (pid > 0)
  {
    printf("Parent process works!\n");
    wait(&status);
  }
  else
    printf("Cannot create child process!\n");
  return 0;
}
```

Функция `wait()` приостанавливает выполнение родительского процесса до тех пор, пока ему не будет послан сигнал, либо пока один из его порожденных процессов не завершится. В операционной системе UNIX имеется и более универсальная функция, `waitpid()`, позволяющая указать, завершения какого именно события следует ожидать.

Если сразу же после запуска программы `process2` выполнить команды, указанные далее, то можно увидеть взаимосвязь родительского и порожденного процессов:

```
bash-3.00# ps -ef|grep UID;ps -ef|grep process2
 UID   PID  PPID   C    STIME TTY         TIME CMD
 root   726   713   0 23:26:01 pts/5       0:00 grep UID
```

```
root   723   722   0 23:25:59 pts/4       0:00 ./process2
root   724   723   0 23:25:59 pts/4       0:00 ./process2
root   728   713   0 23:26:01 pts/5       0:00 grep process2
```

Здесь родительский процесс имеет PID, равный 723, в то время как порожденный процесс имеет PID, равный 724, и PPID (идентификатор родительского процесса), равный 723, как и следовало ожидать.

Пример 3. Здесь иллюстрируется тот факт, что данные, передаваемые порожденному процессу родительским, могут изменяться независимо от данных родительского процесса. Исходный текст программы (она называется `process3`) приведен далее:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
  int i1 = -100;
  int pid, status;
  if ((pid = fork()) == 0)
  {
    i1 = 25;
    printf("Child process: i1 = %d\n", i1);
    return 0;
  }
  else if (pid > 0)
  {
    wait(&status);
    printf("Parent process: i1 = %d\n", i1);
  }
  else
    printf("Cannot create child process!\n");
  return 0;
}
```

Здесь целочисленная переменная `i1` принимает значение –100 в родительском процессе. В порожденном процессе переменной `i1` присвоено значение 25. Поскольку в родительском и порожденном процессах мы имеем дело факти-

чески с разными переменными, то и результаты вывода оператором `printf()` будут разными:

```
bash-3.00# ./process3
Child process: i1 = 25
Parent process: i1 = -100
```

Пример 4. В этом примере показано применение неименованных каналов для взаимодействия родительского и порожденного процессов:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
  int pid;
  int fifo[2], status;
  char buf[128];
  if (pipe(fifo) == -1)
  {
    printf("Error of creating pipe!\n");
    exit(1);
  }
  if ((pid = fork()) == 0)
  {
    char *str = "Child is executed with PID = ";
    int cur_pid = getpid();
    int bytesRead = sprintf(buf, "%s %d", str, cur_pid);
    buf[bytesRead] = 0;
    close(fifo[0]);
    write(fifo[1], buf, bytesRead);
    close(fifo[1]);
    sleep(20);
    exit(0);
  }
  else if (pid > 0)
```

```
  {
    close(fifo[1]);
    bzero(buf, sizeof(buf));
    while (read(fifo[0], buf, sizeof(buf)))
      printf("%s\n", buf);
    close(fifo[0]);
    wait(&status);
    printf("Done.\n");
  }
  else
    printf("Cannot create child process!\n");
  return 0;
}
```

В этой программе родительский процесс создает неименованный канал посредством системного вызова `pipe()` в операторе

```
if (pipe(fifo) == -1)
```

Параметром функции `pipe()` является массив `fifo` из двух целочисленных элементов. Первому элементу `fifo[0]` массива присваивается дескриптор для чтения, а второму, `fifo[1]` — дескриптор для записи. При этом процесс использует дескриптор `fifo[0]` для чтения данных из канала, а дескриптор `fifo[1]` — для записи данных в канал. К хранящимся в канале данным доступ производится по схеме "первым пришел — первым ушел" или FIFO (first in — first out). Кроме того, процесс не может выполнить произвольную выборку данных из канала, начиная с определенной позиции. После считывания данные немедленно удаляются из канала.

С помощью оператора

```
if ((pid = fork()) == 0)
```

создается новый процесс. В этом процессе закрывается дескриптор канала чтения оператором `close(fifo[0])`, оставляя, таким образом, только возможность записи данных.

В родительском процессе, наоборот, закрывается дескриптор канала записи (оператор `close(fifo[1])`), что позволяет процессу выполнять чтение данных из канала.

В порожденном процессе функция `sprintf()` формирует строку, которая записывается в канал с помощью оператора

```
write(fifo[1], buf, bytesRead);
```

использующего системный вызов `write()`. Родительский процесс ожидает данные из канала и читает их с помощью оператора

```
while (read(fifo[0], buf, sizeof(buf)))
 printf("%s\n", buf);
```

После выполнения соответствующих операций дескрипторы каналов должны быть закрыты (операторы `close(fifo[0])` и `close(fifo[1])`).

Пример 5. В данном примере показан еще один метод организации взаимодействия между родительским и дочерними процессами, основанный на применении функции `socketpair()`. Эта функция создает пару неименованных сокетов, для которых указываются тип домена, тип протокола и сам протокол. Оба созданных сокета являются абсолютно идентичными.

Функция имеет синтаксис:

```
int socketpair(int domain, int type, int protocol, int sv[2])
```

Значения дескрипторов помещаются в элементы массива `sv`, причем `sv[0]` содержит дескриптор сокета, предназначенного для чтения данных, а `sv[1]` — дескриптор сокета для записи данных. При использовании этой функции в программе следует применять файлы заголовков

```
#include <sys/types.h>
#include <sys/socket.h>
```

В случае удачного выполнения функция `socketpair()` возвращает 0, в случае неудачи — –1.

Далее приводится исходный текст программы (она называется `socketpair_demo`), в которой дочерний процесс записывает строку данных в канал, а основной процесс читает эти данные из канала. Замечу, что для чтения/записи дескрипторов неименованных сокетов используются, как обычно, системные вызовы `read()` и `write()`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(void)
{
  int sv[2];
```

```
  pid_t pid;
  char *msg_write = "Message from child process";
  char buf[128];
  if (socketpair(AF_UNIX, SOCK_STREAM, 0, sv) == -1)
  {
    printf("Socketpair function error!\n");
    exit(1);
  }
  pid = fork();
  if (pid == 0)
  {
    printf("Child process created...\n");
    close(sv[0]);
    write(sv[1], msg_write, strlen(msg_write));
    close(sv[1]);
    exit(0);
  }
  if (pid > 0)
  {
   printf("Waiting for message from child...\n");
   close(sv[1]);
   int bytesRead = read(sv[0], buf, sizeof(buf));
   if (bytesRead > 0)
   {
     buf[bytesRead] = 0;
     printf(": %s\n", buf);
     close(sv[0]);
   }
 }
 return 0;
}
```

Объясню некоторые ключевые моменты этой программы. Пара неименованных сокетов создается оператором

```
if (socketpair(AF_UNIX, SOCK_STREAM, 0, sv) == -1)
```

Затем системный вызов `fork()` порождает дочерний процесс, который записывает строку `msg_write` в дескриптор `sv[1]` с помощью системного вызова `write()`. Обратите внимание на то, что порожденный процесс может закрыть

при помощи системного вызова `close()` дубликат дескриптора `sv[0]`, если тот ему не нужен (как в данном случае).

В основном процессе используется дескриптор `sv[0]`, из которого данные читаются в буфер `buf` с помощью системного вызова `read()`. Результат работы программы показан далее:

```
bash-3.00# ./socketpair_demo
Waiting for message from child...
Child process created...
: Message from child process
```

Пример 6. Для коммуникаций между отдельными приложениями часто используются именованные каналы (named pipes). Реализация такого метода довольно проста, что показано в следующих примерах.

Вначале следует создать именованный канал с помощью одной из команд: `mknod` или `mkfifo`. Например, для создания именованного канала test.pipe можно воспользоваться одной из команд:

```
bash-3.00# mknod test.pipe p
bash-3.00# mkfifo test.pipe
```

Естественно, что для создания именованного канала пользователь должен обладать правами root.

После этого будет создан именованный канал test.pipe, атрибуты которого можно просмотреть командой `ls -l`:

```
bash-3.00# ls -l test.pipe
prw-r--r--   1 root root 0 Apr 18 12:11 test.pipe
```

Признаком того, что данный объект файловой системы является именованным каналом, служит литера `p` в поле атрибутов доступа.

Созданный таким образом именованный канал может использоваться иными программами для обмена данными друг с другом. Рассмотрим простой пример. Приведенная далее команда пытается прочитать данные из именованного канала test.pipe и, пока их нет, ожидает поступления информации:

```
bash-3.00# cat < test.pipe
```

Если теперь запустить во втором окне другую команду, которая записывает данные в test.pipe, например,

```
bash-3.00# echo Test String for Named Pipe > test.pipe
```

то первая программа эти данные получит:

```
bash-3.00# cat < test.pipe
Test String for Named Pipe
```

Удалить именованный канал test.pipe можно при помощи команды `rm`:

```
bash-3.00# rm test.pipe
```

Для записи данных в созданный ранее именованный канал из программ пользователя можно воспользоваться приведенными далее примерами.

Первый пример позволяет записывать данные в именованный канал с именем test.pipe. Вот исходный текст программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int main(void)
{
  char *buf = "Data will be written into test.pipe\n";
  int fd = open("test.pipe", O_WRONLY);
  write(fd, buf, strlen(buf));
  close(fd);
}
```

Используя исходный текст второго примера, можно скомпилировать приложение, позволяющее прочитать данные из test.pipe:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int main(void)
{
  char buf[128];
  int bytesRead;
  bzero(buf, sizeof(buf));
  int fd = open("test.pipe", O_RDONLY);
  do
  {
```

```
    bytesRead = read(fd, buf, sizeof(buf));
  } while (bytesRead > 0);
  printf("%s\n", buf);
  close(fd);
}
```

Смысл операторов программ достаточно очевиден, поэтому я не буду останавливаться на анализе исходного текста.

Пример 7. Данный пример демонстрирует применение именованных каналов, реализованных в виде сокетов UNIX, для обмена данными между двумя изолированными процессами. Обмен данными между процессами осуществляется с использованием потоковых сокетов (SOCK_STREAM). Один из них (сервер) ожидает поступления данных от другого процесса (клиента). Для упрощения предполагаем, что сервер работает в блокирующем режиме. Сервер запускается программой `uns2`, исходный текст которой находится в файле uns2.c, а клиент реализован в виде программы `unc2` с исходным текстом в файле unc2.c.

Вначале проанализируем работу программы-сервера. Вот исходный текст файла uns2.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(void)
{
  struct sockaddr_un local, client;
  int fdlocal, fdclient;
  char buf[128];
  int isize = sizeof(client);

  unlink("unsocket");
  bzero(&local, sizeof(local));
  strcpy(local.sun_path, "unsocket");
  local.sun_family = PF_UNIX;

  fdlocal = socket(PF_UNIX, SOCK_STREAM, 0);
```

```
  bind(fdlocal, (struct sockaddr*)&local, sizeof(local));
  listen(fdlocal, 20);
  printf("Local Server waits for request...\n");
  while (1)
  {
    fdclient = accept(fdlocal, (struct sockaddr*)&client,
                      (socklen_t*)&isize);
    if (fdclient > 0)
    {
      int ret = recv(fdclient, buf, sizeof(buf), 0);
      if (ret > 0)
      {
        buf[ret] = 0;
        printf(":: %s\n", buf);
      }
    }
    close(fdclient);
  }
  close(fdlocal);
  return 0;
}
```

Если вы не сталкивались с разработкой программ посредством сокетов, то вначале исходный текст может показаться сложным, поэтому для более четкого представления механизма работы сокетов можно обратиться к рис. 7.5.

Вначале необходимо определить переменные структуры типа `sockaddr_un`, которые будут содержать параметры сокетов:

```
struct sockaddr_un local, client;
```

Здесь определены две переменные — `local` и `client`. Переменная `local` будет содержать параметры сокета, на котором сервер будет прослушивать запросы на соединения. Переменная `client` будет содержать параметры сокета для конкретного входящего соединения.

Переменные

```
int fdlocal, fdclient;
```

будут содержать дескрипторы прослушивающего сокета и сокета, на котором принимается конкретное соединение. Предположим, что прослушивающему сокету соответствует имя `unsocket` дискового файла. Перед инициализацией

сервера необходимо удалить файл сокета, что выполняет системный вызов `unlink()`:

```
unlink("unsocket");
```

Далее, структура `local` инициализируется соответствующими значениями:

```
strcpy(local.sun_path, "unsocket");
local.sun_family = PF_UNIX;
```

Здесь поле `sun_path` структуры `local` должно содержать имя прослушивающего сокета, а поле `sun_family` — тип сокета или, как нередко говорят, тип домена. В данном случае тип домена имеет значение `PF_UNIX`. Во многих версиях операционных систем также используется обозначение `AF_UNIX` или `AF_LOCAL`.

Следующий шаг — создание прослушивающего сокета с использованием системного вызова `socket()`:

```
fdlocal = socket(PF_UNIX, SOCK_STREAM, 0);
```

В качестве параметров системного вызова `socket()` задаются тип сокета (PF_UNIX) и тип обмена (SOCK_STREAM). В случае удачного завершения функция возвращает дескриптор `fdlocal` сокета. Затем выполняется привязка дескриптора сокета к структуре `local`, содержащей параметры сокета. Эта операция выполняется при помощи системного вызова `bind()`:

```
bind(fdlocal, (struct sockaddr*)&local, sizeof(local));
```

На следующем шаге сокет устанавливается в режим прослушивания входящих соединений системным вызовом `listen()`:

```
listen(fdlocal, 20);
```

Здесь в качестве первого параметра задается дескриптор сокета, а в качестве второго — максимальное количество соединений, которое одновременно может обработать сервер.

После выполнения этого фрагмента программы сокет, на который указывает дескриптор `fdlocal`, устанавливается в режим прослушивания и готов к обработке соединений, которые могут быть затребованы клиентами.

Далее происходит прием соединений в бесконечном цикле `while (1)` с использованием системного вызова `accept()`:

```
while (1)
{
  fdclient = accept(fdlocal, (struct sockaddr*)&client,
                    (socklen_t*)&isize);
. . .
```

Здесь я хочу обратить внимание читателей на очень важный момент. Функция `accept()` возвращает дескриптор `fdclient` нового сокета, который используется только для данного соединения. Обмен данными с конкретным клиентом будет осуществляться только по этому сокету, при этом прослушивающий сокет с дескриптором `fdlocal` используется только как точка получения запросов на соединение!

В качестве параметров функция `accept()` принимает дескриптор прослушивающего сокета, а также указатель на структуру (`client` в данном случае), содержащую параметры вновь созданного сокета, и адрес переменной, содержащей размер этой структуры.

При удачном завершении системного вызова `accept()` сервер принимает данные от клиента:

```
if (fdclient > 0)
{
  int ret = recv(fdclient, buf, sizeof(buf), 0);
  . . .
```

Прием данных осуществляется путем вызова функции `recv()`, в качестве параметров принимающей дескриптор сокета, указатель на область памяти (в данном случае `buf`), где будут сохраняться данные, размер области данных и флаг (принимаем его равным нулю). Функция `recv()` возвращает число фактически принятых байтов.

Замечу, что в нашем примере предполагается, что буфер памяти для приема данных имеет достаточный размер для приема всего сообщения. Если размер сообщения превышает размер буфера, то можно воспользоваться конструкцией наподобие

```
if (fdclient > 0)
{
  do {
       int ret = recv(fdclient, buf, sizeof(buf), 0);
       if (ret > 0)
       {
         . . .
       }
       . . .
  while (ret > 0);
. . .
```

После окончания приема данных обязательно следует закрыть соединение с клиентом путем закрытия дескриптора `fdclient`:

```
close(fdclient);
```

а при выходе из программы закрыть дескриптор прослушивающего сокета:

```
close(fdlocal);
```

Хочу сделать важное замечание: данный процесс-сервер работает в блокирующем режиме, т. е. установка следующего соединения допускается только после завершения операций обмена данными при установленном текущем соединении. Для одновременной обработки запросов от нескольких клиентов требуются другие подходы, которые здесь рассматриваться не будут. Читатели, которые заинтересуются данной темой, смогут найти достаточно материала как в литературе, так и на интернет-сайтах.

Процесс-клиент, работающий с данным сервером, реализован намного проще, и его исходный текст помещен в файл unc2.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(void)
{
  struct sockaddr_un client;
  int fdclient;
  char *buf = "TEST STRING FOR SERVER";

  bzero(&client, sizeof(client));
  strcpy(client.sun_path, "unsocket");
  client.sun_family = PF_UNIX;

  fdclient = socket(PF_UNIX, SOCK_STREAM, 0);
  connect(fdclient, (struct sockaddr*)&client, sizeof(client));
  send(fdclient, buf, strlen(buf), 0);
  close(fdclient);
  return 0;
}
```

Программа-клиент создает единственный сокет с дескриптором `fdclient`, через который будет выполняться передача данных серверу:

```
fdclient = socket(PF_UNIX, SOCK_STREAM, 0);
```

В отличие от сервера, здесь не требуется привязки адресной структуры `sockaddr_un` к сокету функцией `bind()` — клиент пытается соединиться с указанным сокетом посредством системного вызова `connect()`:

```
connect(fdclient, (struct sockaddr*)&client, sizeof(client));
```

После установки соединения данные передаются при помощи системного вызова `send()`, в качестве параметров которого указывается дескриптор сокета, буфер памяти, где располагаются данные, размер данных и дополнительный флаг.

В данном случае клиент передает строку "`TEST STRING FOR SERVER`", размещенную в буфере памяти `buf`. По окончанию передачи данных дескриптор сокета необходимо закрыть, применив функцию `close(fclient)`.

Для проверки работоспособности программ следует запустить в двух терминальных окнах программу-клиент и программу-сервер. Например, в операционной системе Solaris 5.11 результат работы программ может выглядеть так:

```
bash-3.00# ./uns2
Local Server waits for request...
:: TEST STRING FOR SERVER
:: TEST STRING FOR SERVER
:: TEST STRING FOR SERVER


bash-3.00# ./unc2
bash-3.00# ./unc2
bash-3.00# ./unc2
```

Если вы просмотрите рабочий каталог программы-сервера, то обнаружите там специальный файл unsocket:

```
bash-3.00# ls -l uns*
srwxr-xr-x   1 root     root           0 Feb 16 17:46 unsocket
```

Атрибут `s` говорит о том, что файл unsocket является сокетом.

В операционной системе UNIX информацию о сетевых и локальных соединениях можно получить с помощью утилиты `netstat` с опцией `-a`. Если процесс-сервер `uns2` работает с сокетом `unsocket`, то информация о нем будет выведена на консоль:

```
bash-3.00# netstat -a
Active UNIX domain sockets
Address  Type          Vnode       Conn     Local Addr      Remote Addr
. . .
d60503f8 stream-ord d629dd80      00000000    unsocket
```

В рассмотренном нами примере сервер принимал данные, посланные ему клиентами. Нетрудно написать программу-сервер, которая будет отсылать какие-либо данные в ответ на запрос клиента, который в этом случае будет работать на прием.

Завершая анализ примеров, следует отметить, что в операционной системе UNIX реализовано еще несколько механизмов обмена данными между процессами, таких, например, как потоки, сообщения и разделяемая память, которые сложнее для анализа и здесь рассматриваться не будут.

7.4. Управление процессами из командного интерпретатора shell

Анализ выполняемых в системе процессов — необходимая процедура, с которой рано или поздно сталкивается каждый пользователь, желающий разобраться в работе UNIX и, возможно, оптимизировать ее работу. Операционная система предоставляет для этого целый ряд полезных команд и утилит, позволяющих получать статистику выполняемых процессов, а также управлять приоритетами их выполнения. Наиболее часто используется команда `ps`, присутствующая в тех или иных модификациях во всех операционных системах. С ее помощью можно выяснить, какие программы и как расходуют процессорное время, определить зависшие процессы и даже исследовать производительность системы.

Команда `ps` (от англ. *process status* — состояние процесса) выводит на консоль статус одного или нескольких процессов, выполняющихся в операционной системе. Существуют две различные версии команды `ps`: одна используется в операционных системах, совместимых с System V, другая — в операционных системах, использующих стандарт BSD. В некоторых версиях UNIX могут присутствовать обе реализации команды `ps`. Например, в операционных системах FreeBSD можно обнаружить версию System V команды `ps` в каталоге /usr/5bin. Если вы работаете с системой, совместимой с System V, то реализация команды `ps` для FreeBSD может присутствовать в каталоге /usr/ucb.

Следует сказать, что версии команды `ps` в System V и FreeBSD существенно различаются, поэтому при анализе работы `ps` будет сделан упор на стандарт POSIX, определяющий общие параметры команды для всех операционных систем.

Команда может запускаться как с параметрами, так и без них, причем выполнить ее может любой пользователь системы, не только суперпользователь

root. Заданная без параметров, команда `ps` выводит на консоль список всех процессов. Команда имеет следующий синтаксис:

```
ps [-adelf]
   [-t список_терминалов]
   [-p список_идентификаторов_процессов]
   [-u|U список_идентификаторов_пользователей]
   [-g список_идентификаторов_лидеров_групп]
   [-G список_числовых_идентификаторов_групп]
```

Смысл указанных опций приводится далее:

- `-a` — позволяет вывести информацию обо всех процессах, кроме групповых и не связанных с терминалом;
- `-d` — позволяет вывести информацию обо всех процессах, за исключением лидеров сеанса;
- `-e` — позволяет вывести информацию обо всех процессах;
- `-l` — позволяет сгенерировать длинный листинг отчета;
- `-f` — позволяет сгенерировать полный листинг отчета;
- `-g` *список_идентификаторов_лидеров_групп* — позволяет вывести информацию только о процессах, для которых указаны идентификаторы лидеров групп. Лидер группы — это процесс, номер которого идентичен его идентификатору группы. Например, командный интерпретатор, запускаемый при входе в систему, является стандартным примером лидера группы;
- `-G` *список_числовых_идентификаторов_групп* — позволяет вывести информацию обо всех процессах, имеющих отношение к указанным номерам групп;
- `-p` *список_идентификаторов_процессов* — позволяет вывести информацию только для процессов с указанными идентификаторами;
- `-t` *список_терминалов* — позволяет вывести информацию для процессов, связанных с указанными терминалами;
- `-U` *список_идентификаторов_пользователей* — позволяет вывести информацию обо всех процессах, связанных с указанными идентификаторами пользователей;
- `-u` *список_идентификаторов_пользователей* — позволяет вывести информацию обо всех процессах, связанных с указанными именами пользователей.

Команда `ps` при выводе результата на консоль может устанавливать значения для следующих полей, символические обозначения которых приведены далее:

- `ADDR` — адрес процесса в памяти;

- `C` — доля выделенного планировщиком процессов времени процессора;
- `CMD` — имя команды и аргументы (для опции `-f`);
- `F` — шестнадцатеричные флаги, логическая сумма которых дает следующие сведения о процессе:
  - 00 — процесс завершен; элемент таблицы процессов свободен;
  - 01 — системный процесс: всегда в основной памяти;
  - 02 — процесс трассируется родительским процессом;
  - 04 — родительский трассировочный сигнал остановил процесс;
  - 08 — процесс не может быть разбужен сигналом;
  - 10 — процесс в основной памяти;
  - 20 — процесс в основной памяти; блокирован до завершения события;
  - 40 — идет сигнал к удаленной системе;
  - 80 — процесс в очереди на ввод/вывод;
- `NI` — поправка приоритета;
- `PID` — идентификатор процесса;
- `PPID` — идентификатор родительского процесса;
- `PRI` — текущий приоритет процесса;
- `S` — состояние процесса, зашифрованное одной из аббревиатур:
  - `B`, `W` — процесс находится в состоянии ожидания;
  - `I` — создание процесса;
  - `O` — процесс выполняется;
  - `R` — процесс находится в очереди готовых к выполнению процессов;
  - `S` — процесс не активен;
  - `T` — процесс трассируется;
  - `X` — процесс ожидает выделения дополнительной оперативной памяти;
  - `Z` — процесс "зомби";
- `STIME` — время запуска процесса;
- `SZ` — размер (в блоках по 512 байтов) образа процесса в памяти;
- `TIME` — общее время выполнения для процесса;
- `TTY` — терминальная линия процесса;
- `UID` — идентификатор пользователя владельца процесса;
- `WCHAN` — адрес события, которого ожидает процесс. У активного процесса этот столбец пустой.

В зависимости от указанных опций и реализации с помощью команды ps можно получить и другие атрибуты.

Если ps вводится без параметров, то результат может выглядеть так:

```
bash-3.00# ps
PID TTY          TIME CMD
700 pts/4        0:00 sh
701 pts/4        0:00 bash
702 pts/4        0:00 ps
bash-3.00#
```

Здесь поле PID показывает идентификатор данного процесса, в поле TTY выводится логический терминал, используемый процессом. Если в этом поле указано ?, то процесс не связан ни с какой терминальной линией. В это поле может выводиться console — в этом случае процесс связан с системной консолью.

Если процесс использует так называемый псевдотерминал, т. е. сетевое соединение или соединение с графическим интерфейсом, то в поле TTY отображается аббревиатура, начинающаяся с pt или с p. Для нашего примера все три процесса используют соединение с графическим интерфейсом (pts/4). Поле TIME показывает, сколько процессорного времени израсходовал процесс. Поле COMD (может быть COMMAND или CMD) показывает выполняемую программу.

Как упоминалось ранее, версии команды ps для операционных систем System V и BSD несколько отличаются. На компьютере с выполняющейся операционной системой UNIX могут присутствовать одновременно обе версии ps. В этом случае полезно просмотреть man-страницу для команды ps, чтобы правильно определить опции и интерпретировать результат. Рассмотрим, как работает команда ps в операционных системах System V.

Если ввести в командной строке ps -f, то результат может выглядеть так:

```
bash-3.00# ps -f
UID    PID  PPID   C    STIME TTY          TIME  CMD
root   709   701   0 18:16:18 pts/4       0:00  ps -f
root   700   693   0 18:13:46 pts/4       0:00  sh
root   701   700   0 18:13:50 pts/4       0:00  bash
bash-3.00#
```

Здесь поле UID указывает на пользователя — владельца процесса (root в данном случае). Поле PPID выводит идентификатор родительского процесса, поле C используется планировщиком, а поле STIME показывает время запуска процесса.

При выполнении команды `ps -l` получим такой результат:

```
$ ps -l
F S  UID   PID PPID  C   PRI NI  ADDR      SZ   WCHAN    TTY    TIME CMD
8 S  343  1400 1398  80  1   20  fc315000  125  fc491870 pts/5  0:01 sh
8 O  343  1407 1400  11  1   20  fc491800  114           pts/5  0:00 ps
$
```

Отметим, что `UID` имеет числовое представление. Поле `PRI` показывает приоритет процесса — меньшее числовое значение означает больший приоритет для планировщика. Поле `NI` содержит значение поправки приоритета, а `SZ` показывает размер исполняемого образа процесса. Поле `WCHAN` указывает на событие, которое процесс ожидает. В некоторых операционных системах System V может выполняться планирование процессов в реальном времени, что позволяет использовать опцию `-c` для просмотра работающих процессов:

```
$ ps -c
 PID   CLS PRI  TTY      TIME  COMD
1400   TS  62   pts/5    0:01  sh
1409   TS  62   pts/5    0:00  ps
```

Поле `CLS` показывает класс приоритета процесса. `TS` в данном случае означает работу в режиме с разделением времени, что обычно и происходит. В этом поле может присутствовать значение `SYS` (для системных процессов) или `RT` (процесс выполняется в реальном масштабе времени).

Если вы работаете в операционной системе, совместимой с BSD, то вывод команды `ps` без параметров будет таким, как показано далее:

```
$ ps
  PID  TT  STAT  TIME  COMMAND
22711  co  T     0:00  rlogin brat
22712  co  T     0:00  rlogin brat
23121  co  R     0:00  ps
```

Поле `PID` указывает на идентификатор процесса. Поле `TT` показывает терминал, с которым связан данный процесс. Если в этом поле стоит `?`, процесс не имеет управляющего терминала. Если в этом поле присутствует `co`, процесс выполняется на системной консоли. Терминальные линии могут представлять собой псевдотерминалы, т. е. сетевые соединения или соединения с графическим интерфейсом пользователя. Поле `STAT` показывает состояние процесса. Для более подробной информации об этой опции можно посмотреть man-страницу для команды `ps`. Поле `TIME` сообщает, сколько процессорного времени использует процесс.

Поле COMMAND показывает, какая команда процесса выполняется в данный момент. Обычно здесь можно увидеть список аргументов команды, сохраненных самим процессом. Однако в некоторых системах аргументы команды могут быть изменены самим процессом! Если в команде ps использовать опцию -c, на консоль выводится имя команды без аргументов.

Замечу, что синтаксис команды ps в операционных системах BSD отличается от стандарта UNIX. Команда не использует дефис при указании опций. Выполним команду ps в режиме BSD с опцией l:

```
$ ps l
F        UID PID PPID CP PRI NI SZ  RSS WCHAN    STAT TT  TIME COMMAND
20088201 343 835 834  1  15  0  32  176 kernelma S  p0  0:00 -ksh TERM=vt
20000001 343 861 835  25 31  0  204 440          R  p0  0:00 ps l
20088001 343 857 856  0  3   0  32  344 Heapbase S  p1  0:00 -ksh HOME=/t
```

Поле F показывает флаги, характеризующие состояние процесса. Поля PID и PPID указывают на родительский и порожденный процессы соответственно. Поле UID показывает идентификатор пользователя — владельца процесса. CP предоставляет информацию об использовании процесса для планировщика. Поле PRI содержит значение приоритета процесса: меньшее число означает больший приоритет и наоборот. SZ показывает размер исполняемого образа процесса в 512-байтовых блоках.

Очень важную информацию несет поле RSS — здесь показан фактический размер памяти, занимаемый процессом. WCHAN сообщает, что процесс ожидает наступления события. Интерпретация поля WCHAN является специфической для данной операционной системы.

Посмотрим, что произойдет, если ввести команду ps с параметром u:

```
# ps u
USER     PID  %CPU  %MEM   SZ   RSS TT  STAT  START   TIME  COMMAND
yury   23127  0.0   1.6    200  416 co  R     19:25   0:00  ps u
yury   22712  0.0   0.0    48   0   co  TW    18:40   0:00  rlogin rterm
yury   22711  0.0   0.0    48   0   co  TW    18:40   0:00  rlogin rterm
```

Назначения большинства полей мы знаем, но здесь появились и другие. Поля %CPU и %MEM показывают долю процессорного времени и памяти, используемых процессом. Поле START указывает на момент времени, когда процесс был запущен.

Очень интересным и полезным может оказаться результат выполнения команды ps с опцией v:

```
$ ps v
  PID TT STAT  TIME  SL  RE  PAGEIN  SIZE  RSS   LIM  %CPU  %MEM  COMMAND
```

```
23126 co R     0:00  0   0   0       200   420   xx    0.0  1.6   ps
$
```

Поле SL сообщает, сколько времени процесс находится в состоянии сна, ожидая события. RE позволяет узнать, как долго процесс остается резидентным (постоянно присутствующим) в оперативной памяти. Поле PAGEIN показывает число операций чтнения диска, выполненных процессом, для подкачки страниц, не являющихся резидентными в оперативной памяти. Поле LIM указывает на программное ограничение для используемой памяти.

Пользователь может увидеть все, чем владеет, с помощью команды

```
ps –u идентификатор_пользователя
```

Например, для пользователя root список процессов будет выглядеть так:

```
 # ps –u root
   PID TTY         TIME CMD
    0  ?           0:41 sched
    1  ?           0:00 init
    2  ?           0:00 pageout
    3  ?           0:00 fsflush
  135  ?           0:00 powerd
    7  ?           0:03 svc.star
    9  ?           0:06 svc.conf
  246  console     0:00 ttymon
  131  ?           0:00 nscd
  127  ?           0:00 picld
```

Зная идентификатор процесса, можно отследить его статус. Для систем, совместимых с System V, это можно сделать с помощью команды `ps –p`*PID*:

```
$ ps –p19057
  PID  TTY      TIME COMMAND
19057  ttys3    0:00 ksh
$
```

Для систем, совместимых с BSD, эта же команда выглядит несколько иначе:

```
$ ps l22712
F    UID   PID  PPID CP PRI NI   SZ  RSS WCHAN  STAT TT  TIME COMMAND
8000 343 22712 22711  0   1  0  48    0 socket TW   co  0:00 rlogin rterm
```

При помощи команды ps можно определить статус группы процессов, что может понадобиться при отслеживании отдельного выполняющегося зада-

ния. Для систем, сместимых с System V, для этих же целей можно использовать команду `ps -g`*`PGID`*:

```
$ ps -lg19080
F S  UID   PID  PPID C PRI NI    ADDR SZ    WCHAN TTY   TIME COMD
1 S  343 19080 19057 0 158 24  710340 51   39f040 ttys3 0:58 analysis
1 S  343 19100 19080 0 168 24  71f2c0 87 7ffe6000 ttys3 2:16 market
```

В BSD-системах стандартного способа для получения информации о группе процессов не существует, хотя команда `ps j` предоставляет некоторые полезные сведения:

```
$ ps j
PPID PID PGID  SID TT TPGID STAT UID TIME COMMAND
834  835  835  835 p0   904  SOE 198 0:00 -ksh TERM=vt100 HOME=/home/yury
835  880  880  835 p0   904  TWE 198 0:00 vi
835  881  881  835 p0   904  TWE 198 0:00 vi t1.sh
835  896  896  835 p0   904  IWE 198 0:00 ksh t2.sh _ /usr/local/bin/k
896  897  896  835 p0   904  IWE 198 0:00 task_a
896  898  896  835 p0   904  IWE 198 0:00 task_b
835  904  904  835 p0   904   RE 198 0:00 ps j
```

Обратите внимание на поле `PGID` для идентификаторов процессов `PID`, находящихся в диапазоне 896—898. Обратите также внимание на поле `TPGID`, оно одинаковое для всех процессов и указывает на владельца терминала.

Анализ статуса отдельной терминальной линии может стать полезным для фильтрации процессов, запущенных под отдельной учетной записью либо с терминала, либо через сеть. Для систем, совместимых с System V, используется синтаксис

```
ps -t идентификатор_терминала
```

Пример использования такого варианта команды приведен далее:

```
$ ps -fts3
UID    PID  PPID  C    STIME TTY      TIME COMMAND
root 19056   136  0 19:21:00 ttys3    0:00 rlogind
yury 19080 19057  0 19:23:53 ttys3    1:01 analysis
yury 19057 19056  0 19:21:01 ttys3    0:00 -ksh
yury 19100 19080  0 19:33:53 ttys3    3:43 market
yury 19082 19057  0 19:23:58 ttys3    0:00 vi test
```

В BSD-совместимых системах можно задавать команду

```
ps t идентификатор_терминала
```

что позволяет увидеть процессы, выполняющиеся на отдельном терминале:

```
$ ps utp5
USER       PID  %CPU %MEM    SZ   RSS TT STAT  TIME COMMAND
root      2058   0.0  0.9   286      p5 R     0:00 -sh (sh)
root      2060   0.0  2.7    53      p5 R     0:00 vi crazy.txt
```

Очень часто команда `ps` применяется в конвейерах с другими командами, позволяя получить вывод нужных полей на консоль:

```
$ ps -ef|grep ps
root   3423    1  0 17:03 ?      00:00:00 gpm -t imps2 -m /dev/mouse
root   3443    1  0 17:03 ?      00:00:00 cupsd
yury   3670    1  0 17:06 ?      00:00:00 eggcups --sm-client-id default6
yury   4108 3705  0 17:49 pts/0  00:00:00 ps -ef
yury   4109 3705  0 17:49 pts/0  00:00:00 grep ps
```

Следующая командная строка позволяет получить только идентификаторы процессов PID:

```
$ ps -ef|grep ps|cut -c11-15
3423
3443
3670
4179
4180
```

Поскольку в операционной системе выполняется очень много процессов, чрезвычайно важно обеспечить оптимальную очередность их выполнения. Это может быть достигнуто путем присвоения выполняемому процессу определенного приоритета.

Каждый процесс в операционной системе UNIX выполняется с вполне определенным приоритетом. Планирование приоритетов процессов лежит в основе функционирования операционной системы, а само планирование выполняет так называемый планировщик процессов. Планировщик процессов представляет собой функцию ядра и в процессе функционирования всегда выбирает процесс с наивысшим приоритетом для выполнения. Следует учитывать, что приоритет процесса не является фиксированным и динамически изменяется системой в зависимости от использования вычислительных ресурсов, времени ожидания запуска и текущего состояния процесса. Если процесс готов к запуску и имеет наивысший приоритет, планировщик приостановит выполнение текущего процесса с более низким приоритетом, даже если последний не использовал отведенное ему время.

Ядро операционной системы UNIX функционирует в непрерываемом (non-preemptive) режиме. Это означает, что процесс, выполняющийся в режиме ядра (в результате системного вызова или прерывания) и выполняющий системные инструкции, не может быть прерван системой, а вычислительные ресурсы переданы другому высокоприоритетному процессу. В этом состоянии выполняющийся процесс не может инициировать освобождение процессора и перейти в состояние сна. В противном случае система может прервать выполнение процесса только при переходе из режима ядра в пользовательский режим. Такой подход значительно упрощает решение задач синхронизации и поддержки целостности структур данных ядра.

Любой процесс имеет текущий приоритет, на основании которого происходит планирование, и относительный приоритет или поправку приоритета (nice number), который задается при порождении процесса и влияет на текущий приоритет.

Верхняя и нижняя границы текущего приоритета могут значительно отличаться в различных версиях UNIX и для разных планировщиков. В любом случае процессы, выполняющиеся в пользовательском режиме, имеют более низкий приоритет, чем работающие в режиме ядра.

Во всех операционных системах UNIX пользователи могут при запуске процесса задавать значение поправки приоритета с помощью команды `nice`, которая имеет следующий синтаксис:

`nice [-`*инкремент* `| -n [-|+]`*инкремент*`] `*команда* `[`*аргумент*`...]`

Диапазон значений инкремента в большинстве систем варьируется от –20 до 20. Если инкремент не задан, используется стандартное значение 10. Положительный инкремент означает снижение текущего приоритета. Обычные пользователи могут задавать только положительный инкремент и, тем самым, только снижать приоритет.

Несмотря на различия в диапазонах значений `nice`, смысл опций этой команды во всех операционных системах одинаков. Например, порожденный процесс наследует приоритет своего родительского процесса. В этом случае владелец процесса может увеличить значение `nice`, но уменьшить его он не может, что препятсятвует процессам с низким приоритетом порождать дочерние процессы с более высоким приоритетом. Имеется единственное исключение из этого правила, позволяющее суперпользователю root устанавливать значение приоритета командой `nice`. Может сложиться ситуация, когда присвоение очень высокого приоритета какому-нибудь процессу приведет практически к остановке всех остальных процессов.

Пользователь root может задать отрицательный инкремент, который повышает приоритет процесса и, тем самым, способствует его более быстрой работе:

```
# nice -n -10 user_proc
```

Здесь приоритет процесса `user_proc` будет повышен.

Приоритет процесса устанавливается в момент его создания командой `nice`. Изменить значение приоритета можно во время выполнения процесса при помощи команды `renice`. Эта команда в качестве первого параметра принимает поправку приоритета, а вторым параметром является идентификатор процесса:

```
# renice -9 7412
```

Здесь повышается приоритет процесса с идентификатором 7412. Следует сказать, что единого соглашения о способе задания приоритета в различных операционных системах не существует, поэтому в случае необходимости лучше всего руководствоваться man-страницами.

Во многих случаях процессы необходимо запускать в конкретное время или через определенные интервалы времени. Необходимость в планировании заданий обычно возникает, когда пользователю нужно выполнить служебные процедуры, требующие длительного времени и использующие значительные системные ресурсы. Так, например, следует периодически проверять свободный объем дискового пространства или проводить архивацию важных данных. Нередко в очередь планируемых заданий помещаются так называемые процедуры сборки "мусора" (удаление временных файлов, очистка системных журналов, удаление процессов "зомби" и др.).

Любой квалифицированный пользователь, как правило, в состоянии разработать собственные процедуры, выполняющиеся в определенные интервалы времени, хотя в операционной системе UNIX для этой цели предусмотрен удобный механизм планирования заданий, позволяющий запускать процессы и задачи в конкретное время или по определенному графику.

При планировании часто употребляют термин "задание", под которым понимают любую команду, введенную из командной строки. Управление заданиями представляет собой механизм, управляющий процессами, выполняемыми в текущем сеансе работы с терминалом. Следует сказать, что в принципе не существует ограничения на число одновременно выполняемых заданий, при этом сами задания могут находиться в состоянии выполнения, установки или пребывать в некотором другом состоянии.

Задание может включать в себя один или несколько командных файлов или одну или несколько команд. В простейшем случае, который встречается чаще всего, задание состоит из одной команды и выполняется как один процесс.

В дальнейшем, если не будет оговорено особо, термины "процесс" и "задание" будут использоваться как синонимы.

Рассмотрим команду `at` — одно из мощных программных средств операционной системы UNIX, которое предназначено для запуска задач в определенное время. Кроме того, эта команда позволяет также управлять и очередью задач.

Командный файл `at` предоставляет механизм для выполнения задач в определенные моменты времени, указав системе, что и когда хотим сделать. Задача остается "спящей" в фоновом режиме до назначенного времени, что предоставляет нам возможность превратить компьютер в будильник, секретаря, администратора встреч и т. д.

Команда `at` имеет следующий синтаксис:

`at <`*опции*`> <`*время*`>`

Проанализируем смысл опций, используемых во всех операционных системах:

- `-f` *файл* — определяет путь к файлу, содержащему задачу;
- `-l` — позволяет вывести список заданий в очереди;
- `-m` — после выполнения задания пользователю будет послано сообщение по электронной почте;
- `-q` — позволяет задать имя очереди;
- `-r` *задание* — позволяет удалить задание из очереди команды `at`.

Если при вызове команды `at` не задана опция `-f`, то для создания задачи используются данные, полученные из стандартного потока ввода. Кроме этих команда `at` может использовать и дополнительные опции, специфичные для каждой операционной системы.

В большинстве случаев при вызове команды `at` указывается единственный параметр — *время*, на которое назначено выполнение задания. Параметр *время* может задаваться в разных форматах.

Обычно время выполнения задается в виде одной, двух или четырех цифр. Одно- и двухзначные числа означают часы, а четырехзначная величина определяет часы и минуты. При задании этого параметра разрешается разделять часы и минуты двоеточием, например, 12:15. В большинстве реализаций операционных систем существуют специальные символьные обозначения для определенных периодов времени. Например, для выполнения задания в полночь можно использовать параметр `midnight`, для запуска задания в полдень — параметр `noon`, а если требуется запустить задание немедленно, можно указать параметр `now`.

Вот примеры задания команды `at`:

```
at 0815am Jan 24
at 8 :15amjan24
at now "+ 1day"
at 5 pm FRIday
at '17
```

Более подробную информацию читатель найдет в man-странице для команды `at`. Следует отметить еще одну особенность выполнения заданий: поскольку команды выполняются в контексте отдельной командной оболочки и отдельной группы процессов вне связи с управляющим терминалом, то никакие открытые файловые дескрипторы и приоритеты не наследуются от вызывающей командной оболочки.

Рассмотрим пример. Предположим, необходимо к определенному моменту времени (пусть это будет в 20:35) знать, какое количество пользователей находится в системе, чтобы выполнить какие-либо действия. В этом случае можно выполнить команду `at` с параметрами, которые указаны далее. Из приглашения `at>` можно выполнить команду `echo` с выводом результата в файл usrlogged. После ввода всех команд задания команда `at` вернет номер задания — уникальный идентификатор, который можно использовать позже для идентификации задания. Когда выполнение команды `at` завершается, в каталоге atjobs системного каталога spool будет создан файл задания. Теперь остается подождать результата выполнения задания. Все описанные шаги показаны далее:

```
# at 20:35
at> echo "Number of users logged in just time = `who|wc -l`" > usrlogged
at> Ctrl-D
job 9 at 2005-01-14 20:35
```

По истечении требуемого интервала времени для выполнения задания в каталоге появится файл usrlogged:

```
# ls -l | grep usrlogged
-rw-r--r--    1 root     root           46 Jan 14 20:35 usrlogged
```

Содержимое файла usrlogged может выглядеть примерно так:

```
# cat usrlogged
Number of users logged in just time = 4
```

В команде `at` время выполнения можно указать в достаточно свободной форме, в данном случае оно задано в формате `чч:мм`. Если нежелательно, чтобы задание было запущено в ближайшие 24 часа, то, кроме времени,

необходимо задать еще и дату запуска. При этом дата должна следовать после времени.

Просмотреть список заданий можно командой `at` с параметром `-l`. Формат вывода зависит от используемой версии операционной системы, однако в любом случае будут отображены дата запуска, имя владельца задания и идентификатор задания, как в этом примере:

```
# at -l
4       2004-08-14 17:01 a root
11      2004-08-14 17:35 a root
```

Кроме команды `at` в операционной системе имеется команда `batch`, позволяющая запустить одну или несколько команд в нужное время. Команда `batch` не имеет параметров, поэтому для ее запуска следует просто ввести ее имя с консоли, затем, как и в команде `at`, ввести список параметров.

Права доступа к командам `at` и `batch` определяются содержимым двух файлов, находящихся в каталоге /usr/lib/cron, — at.allow и at.deny. Если файл at.allow существует, то к программам `at` и `batch` могут обращаться только те пользователи, идентификаторы которых перечислены в этом файле.

Если существует файл at.deny, то у пользователей, идентификаторы которых указаны в этом файле, прав доступа к `at` и `batch` нет. Редактирование этих файлов должен выполнять только суперпользователь root.

7.5. Сигналы

Сигналы обеспечивают механизм вызова определенной функции при наступлении некоторого события и в этом смысле напоминают аппаратные прерывания. Сигналы представляют один из способов межпроцессных коммуникаций и могут оказать неоценимую помощь в отладке процессов. Кроме того, управление процессами с помощью сигналов — это практически единственный метод, предоставляемый командным интерпретатором shell пользователю.

В основе функционирования механизма передачи сигналов лежит понятие события. Каждое событие имеет свой числовой идентификатор, находящийся в диапазоне от 1 до 36, и соответствующее этому идентификатору имя в виде строки символов. Сигнал отправляется при наступлении определенного события, о котором должен быть уведомлен процесс. Сигнал считается доставленным, если процесс, которому он предназначен, получит и выполнит его обработку.

Сигнал может послать любой выполняющийся процесс любому другому процессу. Доставка сигнала возможна в том случае, если оба процесса (пере-

дающий сигнал и принимающий его) принадлежат одному владельцу, или если сигнал послан суперпользователем root. Кроме того, сигнал любому процессу может послать ядро в ответ на ряд событий, которые могут быть вызваны самим процессом, другим процессом, прерыванием или каким-либо внешним событием.

Каждому сигналу в ядре операционной системы соответствует функция обработки по умолчанию, которая выполняется в том случае, если процесс не указал другого действия. Наиболее часто функции — обработчики сигналов выполняют стандартные действия: завершение процесса, игнорирование сигнала, остановку и продолжение выполнения процесса.

Обработчик сигнала предполагает, что процесс все еще выполняется, что может привести к некоторым задержкам между отправкой и доставкой сигнала в системах с высокой загрузкой. Это вызвано тем, что процесс не может получить сигнал, пока не будет выбран планировщиком. Посылка сигналов может быть инициирована и в случае возникновения исключительных ситуаций, например, при выполнении процессом определенной инструкции, вызывающей в системе ошибку. Если такое происходит, вызывается системный обработчик исключительной ситуации, и процесс продолжает выполняться в режиме ядра, причем почти так же, как и при обработке любого другого прерывания. Функция-обработчик отправляет процессу соответствующий сигнал, который доставляется, когда процесс возвращается в пользовательский режим.

Очень важной особенностью механизма отправки и получения сигналов является то, что процесс может посылать сигнал сам себе. Эта возможность часто используется для отладки командных и исполняемых файлов. Посылка сигналов самому себе полезна в тех случаях, когда по каким-то причинам процесс не может завершить работу самостоятельно.

Таким образом, процессу или группе процессов отправляется сигнал в случаях:

- возникновения исключительной ситуации, например, деления на 0;
- возникновения терминального прерывания, вызванного, скажем, нажатием определенных комбинаций клавиш терминала, например, <Del>, <Ctrl>+<C>, <Ctrl>+<\>, что приводит к посылке сигнала текущему процессу, связанному с терминалом;
- инициации со стороны других процессов: процесс может посылать сигнал другому процессу или группе процессов, используя системный вызов `kill`.

С помощью команды `kill -l` можно получить список сигналов, поддерживаемых системой:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
```

```
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      33) SIGRTMIN    34) SIGRTMIN+1
35) SIGRTMIN+2  36) SIGRTMIN+3  37) SIGRTMIN+4  38) SIGRTMIN+5
39) SIGRTMIN+6  40) SIGRTMIN+7  41) SIGRTMIN+8  42) SIGRTMIN+9
43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13
47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-14 50) SIGRTMAX-13
51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10 54) SIGRTMAX-9
55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6  58) SIGRTMAX-5
59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2  62) SIGRTMAX-1
63) SIGRTMAX
```

В приведенном списке сигналов можно выделить те, которые используются наиболее часто:

- `SIGTERM` — требует завершения процесса. Это стандартный сигнал, посылаемый процессу в случае необходимости его завершения;
- `SIGHUP` — требует завершения процесса в случае, если потеряна связь с терминалом или терминальное окно закрыто. В этом случае сигнал посылается всем нефоновым процессам, связанным с соответствующей терминальной линией;
- `SIGKILL` — требует завершения процесса. В отличие от других сигналов, вызывающих завершение, этот является неперехватываемым, что позволяет безусловно завершить любой процесс;
- `SIGILL` — требует завершения процесса и создания дампа (образа) памяти. Посылается ядром в том случае, если центральный процессор обнаружил запрещенную инструкцию, что могло быть следствием недопустимого перехода в машинном коде программы, например, попытки выполнить строку данных;
- `SIGTRAP` — требует завершения процесса и создания дампа памяти. Посылка сигнала инициируется системным вызовом `ptrace`, который используется для отладки, если была установлена ловушка точки прерывания процесса;
- `SIGFPE` — требует завершения процесса и создания дампа памяти. Посылка сигнала инициируется в том случае, если была попытка выполнить за-

прещенную арифметическую операцию, например, взятие логарифма отрицательного числа или деление на 0;

- `SIGBUS` — требует завершения процесса и создания дампа памяти. Посылка сигнала инициируется при возникновении ошибки на шине ввода/вывода, что в большинстве случаев является результатом попытки выполнить чтение или запись вне границ памяти программы;
- `SIGSEGV` — требует завершения процесса и создания дампа памяти. Посылка сигнала инициируется в случае нарушения сегментации памяти;
- `SIGPIPE` — требует завершения процесса в том случае, когда он попытался выполнить чтение или запись в программный канал, который был уже закрыт. Этот сигнал позволяет завершить работу конвейера, когда одна из его команд дала сбой;
- `SIGALRM` — позволяет выполнить конкретные действия по истечении определенного интервала времени.

Среди этих сигналов, в свою очередь, чаще других используются 1 (`SIGHUP`), 2 (`SIGINT`), 15 (`SIGTERM`), а также пользовательские сигналы `SIGUSR1` и `SIGUSR2`. Безусловно уничтожить процесс в большинстве случаев можно сигналом 9 (`SIGKILL`), хотя следует заметить, что в некоторых ситуациях процесс невозможно уничтожить даже сигналом `SIGKILL`!

Из результата выполнения команды `kill -l` видно, что каждому сигналу соответствует определенное целое положительное число, хотя в разных операционных системах эти числа могут отличаться. Тем не менее, основные сигналы, как правило, имеют одно и то же числовое значение в разных системах. Это относится к сигналам `SIGKILL`, `SIGTERM`, `SIGHUP`, `SIGINT`.

В практическом аспекте для посылки процессу сигнала используется команда `kill`, имеющая следующий синтаксис:

```
kill -s [сигнал]  [PID]
```

Здесь `PID` — идентификатор процесса, которому посылается сигнал.

Процесс, принимающий сигнал, во многих случаях должен реагировать на него определенным образом, отличным от того, что предлагают стандартные обработчики сигналов, присутствующие в системе. При этом могут выполняться, например, файловые операции или освобождение занятых ресурсов. Для отслеживания и реакции на приходящие сигналы служит команда `trap`. Она позволяет перехватывать сигнал, посланный процессу, и выполнять определенные действия. В качестве входных параметров этой команде передается список сигналов и командная строка, которую нужно выполнить в случае получения сигнала. Строка может представлять собой набор команд или вызов функции.

Рассмотрим примеры применения сигналов для управления процессами. Для перехвата сигналов с целью их обработки в командный интерпретатор shell включена команда `trap`. Она имеет синтаксис:

```
trap [ argument n1 n2 ... ]
```

Здесь параметр *argument* указывает на функцию или последовательность функций, которые необходимо выполнить при получении одного из сигналов с номером *n1*, *n2* и т. д. Обработка сигналов проводится в той последовательности, в которой они указаны в списке аргументов. Если `argument` является нулевой строкой, то перечисленные в списке сигналы будут проигнорированы командным интерпретатором.

Пример 1.

```
function lsfunc() {
  echo
  echo "Result of executing ls -l:"
  echo
  ls -l
}
xTrap=$$
echo $xTrap > PID
trap "lsfunc" 1 2 15
until false
do
 sleep 1
done
```

Программа находится в бесконечном цикле, ожидая поступления одного из сигналов `SIGHUP` (1), `SIGINT` (2) или `SIGTERM` (15). В момент поступления сигнала вызывается функция `lsfunc`, которая установлена командой `trap` в качестве обработчика сигналов 1, 2 и 15:

```
trap "lsfunc" 1 2 15
```

С помощью функции `lsfunc` выполняется вывод на консоль содержимого текущего каталога по команде `ls -l`. Команда

```
echo $xTrap > PID
```

сохраняет PID процесса в файле PID — это облегчает задачу посылки сигнала процессу. Для активизации функции `lsfunc` в окне терминала наберите одну из команд

```
kill -1 `cat PID`
kill -2 `cat PID`
kill -15 `cat PID`
```

после чего на консоль будет выведено содержимое текущего каталога:

```
Result of executing ls -l:

total 12
-rw-r--r--   1 root     root           5 Feb 19 12:37 PID
-rw-r--r--   1 root     root        1222 Feb 18 19:03 PROCESSES
-rw-r--r--   1 root     root        1800 Feb 19 12:37 TUTORIAL
-rwxr-xr-x   1 root     root         158 Feb 19 12:37 trap_demo1
```

Уничтожить процесс можно командой

```
kill -SIGKILL `cat PID`
```

или

```
kill -9 `cat PID`
```

Пример 2. Процесс может принимать сигналы не только от других процессов, но и посылать сигналы самому себе. Такая возможность очень часто используется для перезапуска и инициализации самого процесса, например, при необходимости перезапуска системных сервисов. Командный файл, исходный текст которого показан далее, — это модифицированный вариант примера 1 с измененным кодом функции `lsfunc`, в который добавлена строка:

```
kill -SIGKILL `cat PID`
```

Вот так выглядит модифицированный вариант командного файла из примера 1:

```
function lsfunc() {
  echo
  echo "Result of executing ls -l:"
  echo
  ls -l
  kill -SIGKILL `cat PID`
}

xTrap=$$
echo $xTrap > PID
trap "lsfunc" 1 2 15
until false
  do
    sleep 1
  done
```

В момент приема одного из сигналов 1, 2 или 15 на консоль выводится содержимое каталога, после чего функция `lsfunc` отправляет текущему процессу сигнал `SIGKILL`, приводящий к его уничтожению.

Пример 3.

```
function fileexists() {
file="new"
if test  -f $file
 then
  rm $file
  echo "File new exists. Program is terminating..."
  kill -SIGKILL `cat PID`
else
  echo "File not found"
fi
}

xTrap=$$
echo $xTrap > PID
trap "fileexists" 1 2 14 15
until false
  do
    sleep 30
    kill -SIGALRM `cat PID`
  done
```

В этом примере команда `trap` добавляет к уже существующим сигналам сигнал `SIGALRM` (14), выполняющий функцию будильника. Каждые 30 секунд функция `fileexists` проверяет наличие на диске файла с именем new. Программный код функции проверяет, существует ли файл new в текущем каталоге, и если это так, удаляет его и завершает текущий процесс посылкой ему сигнала `SIGKILL`. В противном случае происходит завершение функции, и процесс продолжает ожидать создания файла до следующей активизации будильника.

Пример 4. Командный файл, исходный текст которого приведен далее, показывает, как можно безболезненно уничтожить процесс, принимая в качестве аргумента командной строки идентификатор уничтожаемого процесса:

```
for sig in TERM INT HUP QUIT KILL
do
  sleeping=0
  for pid in $*
```

```
    do
      if kill -0 $pid
      then
        kill -$sig $pid
        sleeping=1
      fi
    done
  if [ $sleeping -eq 1 ]
  then
    sleep 1
  fi
done
```

Здесь строка

```
for sig in TERM INT HUP QUIT KILL
```

определяет порядок обработки сигналов `SIGTERM`, `SIGINT`, `SIGHUP`, `SIGQUIT` и `SIGKILL`. Анализ приходящих сигналов выполняется в таком порядке:

1. Первым проверяется сигнал `SIGTERM`, поскольку по умолчанию он уничтожает процесс.
2. Далее проверяется сигнал `SIGINT`, поскольку это обычный способ завершения работы программы.
3. Далее анализируется `SIGHUP`, поскольку многие программы создают файл восстановления.
4. `SIGQUIT` обрабатывается следующим, т. к. он может быть перехвачен программой.
5. Сигнал `SIGKILL` стоит последним в списке обрабатываемых сигналов, поскольку он не может быть ни перехвачен, ни блокирован, ни проигнорирован.

Команда

```
if kill -0 $pid
```

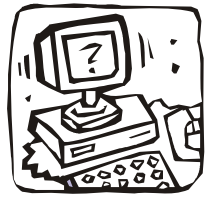
проверяет, выполняется ли еще процесс с указанным PID. Если выполняется, то делается попытка его уничтожить, используя текущий сигнал:

```
kill -$sig $pid
```

После этого выдерживается пауза, чтобы дать процессу время переключиться на обработку следующего сигнала:

```
if [ $sleeping -eq 1 ]
```

Рассмотренные примеры далеко не исчерпывают всех возможностей механизма обработки сигналов операционной системой UNIX, позволяющего строить довольно сложные схемы обмена данными между процессами.



Глава 8

Поддержка сетей в UNIX

Любой пользователь, независимо от того, работает он на домашнем компьютере или на рабочей станции, расположенной на предприятии, так или иначе, столкнется с необходимостью настройки сетевых параметров операционной системы UNIX. И здесь проявляется вся мощь этой операционной системы, особенно учитывая то, что UNIX была задумана как сетевая операционная система. Для лучшего понимания механизмов работы UNIX в сети вначале рассмотрим некоторые теоретические аспекты сетевого взаимодействия.

В самых ранних вариантах UNIX коммуникации осуществлялись с использованием посимвольного обмена данными из-за ограниченности выбора аппаратных средств (в основном использовались модемы и терминалы). Несколько позже в системе появилась поддержка более развитых устройств, протоколов, операционных режимов и т. д., но программные средства по-прежнему основывались на ограниченных возможностях символьного ввода/вывода.

С появлением многоуровневых сетевых протоколов, таких как TCP/IP, SNA, OSI, X.25 и др., стало очевидно, что для операционных систем UNIX требуется общая основа организации сетевого обмена данными, основанная на многоуровневых протоколах. Для решения этой проблемы было реализовано несколько механизмов, обладающих примерно одинаковыми возможностями, но несовместимых между собой, поскольку каждый из них являлся результатом некоторого отдельного проекта. Эту проблему удалось решить путем реализации так называемого механизма потоков (STREAMS), обеспечивающего широкие возможности для создания драйверов устройств и коммуникационных протоколов. Простейшими примерами потоковых модулей являются разного рода перекодировщики символьной информации. Более сложным примером может служить потоковый модуль, осуществляющий разборку нисходящих данных в пакеты для их передачи по сети и сборку восходящих данных с удалением служебной информации пакетов.

Для разрешения проблемы различий в форматах кадров, используемых в разных сетях, был реализован универсальный формат пакета данных, известный как IP-дейтаграмма (Internet Protocol Datagram), реализованный в рамках стандарта протокола IP (Internet Protocol). Такой пакет состоит из заголовка и порции данных, при этом порция данных IP-дейтаграммы содержится внутри сетевого кадра. Это позволяет поместить IP-дейтаграмму в сетевой кадр конкретного формата и передавать далее ее в разных сетях Интернета. При этом все узлы, шлюзы и сети Интернета должны понимать IP-дейтаграммы.

Следует сказать, что дейтаграммы не обрабатываются в каком-либо жестко установленном порядке, напротив, каждая из них обрабатывается независимо от остальных, что позволяет эффективно использовать ресурсы всех связанных между собой узлов сети. К сожалению, сервис, предоставляемый протоколом IP, не является надежным, поскольку не гарантирует доставку пакетов в нужном порядке, отсутствия потерь дейтаграмм или их дублирования.

Данная проблема во многом была решена в рамках протокола TCP (Transmission Control Protocol), обеспечивающего надежную доставку сообщений за счет подтверждений доставки дейтаграмм и их повторной передачи в случае необходимости. Здесь был реализован следующий механизм: если узел, посылающий дейтаграмму, не получает подтверждение о доставке в течение установленного промежутка времени, то дейтаграмма считается недоставленной, и она отправляется повторно. Именно этот механизм и применяется в настоящее время в наиболее широко используемом семействе протоколов TCP/IP (Transmission Control Protocol/Internet Protocol). Семейство, или по-другому, стек протоколов TCP/IP обеспечивает возможности сосуществования компьютерных сетей, основанных на разных технологиях. Бóльшая часть коммуникационных средств ОС UNIX основана на использовании протоколов стека TCP/IP, поэтому проанализируем особенности данного семейства протоколов детально.

На программном уровне протоколы TCP/IP реализованы в виде сокетов (от англ. *socket* — гнездо). Механизм сокетов впервые был реализован в 1982 году в UNIX BSD 4.1 в качестве развитого средства межпроцессных взаимодействий. Это средство позволяет любому процессу обмениваться сообщениями с любым другим процессом, независимо от того, выполняются они на одном компьютере или на разных машинах, соединенных сетью. Мы рассматривали сокеты UNIX в *главе 7*, когда анализировали взаимодействие процессов на локальной машине.

Механизм сокетов является обязательным компонентом всех версий операционных систем UNIX, однако в разных системах он реализован по-разному. В BSD-совместимых системах сокеты реализованы как функции ядра,

и пользователи могут выполнять специальные системные вызовы `socket()`, `bind()`, `listen()`, `connect()` и `accept()` при разработке приложений.

В операционных системах, совместимых с System V, механизм сокетов реализован не в ядре системы, а представлен набором функций в библиотеке /usr/lib/libsocket.a или /usr/lib/libsocket.so.

Стек протоколов TCP/IP обычно изображают в виде семиуровневой модели ISO/OSI, которую можно представить следующими уровнями функциональности (рис. 8.1).

Цель модели — в обеспечении стандарта сетевой связи компьютеров независимо от производителя оборудования компьютеров и/или сети. Здесь следует уточнить термин "сеть". В протоколах сетевого уровня термин "сеть" означает совокупность компьютеров, соединенных между собой в соответствии с одной из стандартных типовых топологий и использующих для передачи пакетов общую базовую сетевую технологию. Внутри сети сегменты не разделяются маршрутизаторами, иначе это была бы не одна сеть, а несколько сетей. Маршрутизатор соединяет несколько сетей в интерсеть.

Все функции физического, канального и сетевого уровней тесно связаны с используемым в данной сети оборудованием: сетевыми адаптерами, концентраторами, мостами, коммутаторами и маршрутизаторами. Функции прикладного, сеансового и уровня представления реализуются операционными системами и системными приложениями конечных узлов. При этом транспортный уровень выступает посредником между этими двумя группами уровней.

В качестве адресов отправителя и получателя в составной сети используется не аппаратный адрес (`MAC`-адрес), а пара чисел — номер сети и номер компьютера в данной сети. В протоколах канального уровня поле "номер сети" обычно отсутствует — предполагается, что все узлы принадлежат одной сети. Явная нумерация сетей позволяет протоколам сетевого уровня составлять точную карту межсетевых связей и выбирать рациональные маршруты при любой их топологии, используя альтернативные маршруты, если они имеются, что невозможно сделать с помощью мостов.

Таким образом, внутри сети доставка сообщений регулируется канальным уровнем, а доставка пакетов между сетями выполняется на сетевом уровне. Рассмотрим смысл уровней модели OSI более детально.

- **Физический уровень.** На этом уровне выполняется передача битов информации по физическим каналам, таким, например, как коаксиальный кабель, витая пара или оптоволоконный кабель. Физический уровень определяет характеристики физических сред передачи данных, а также параметры электрических сигналов.

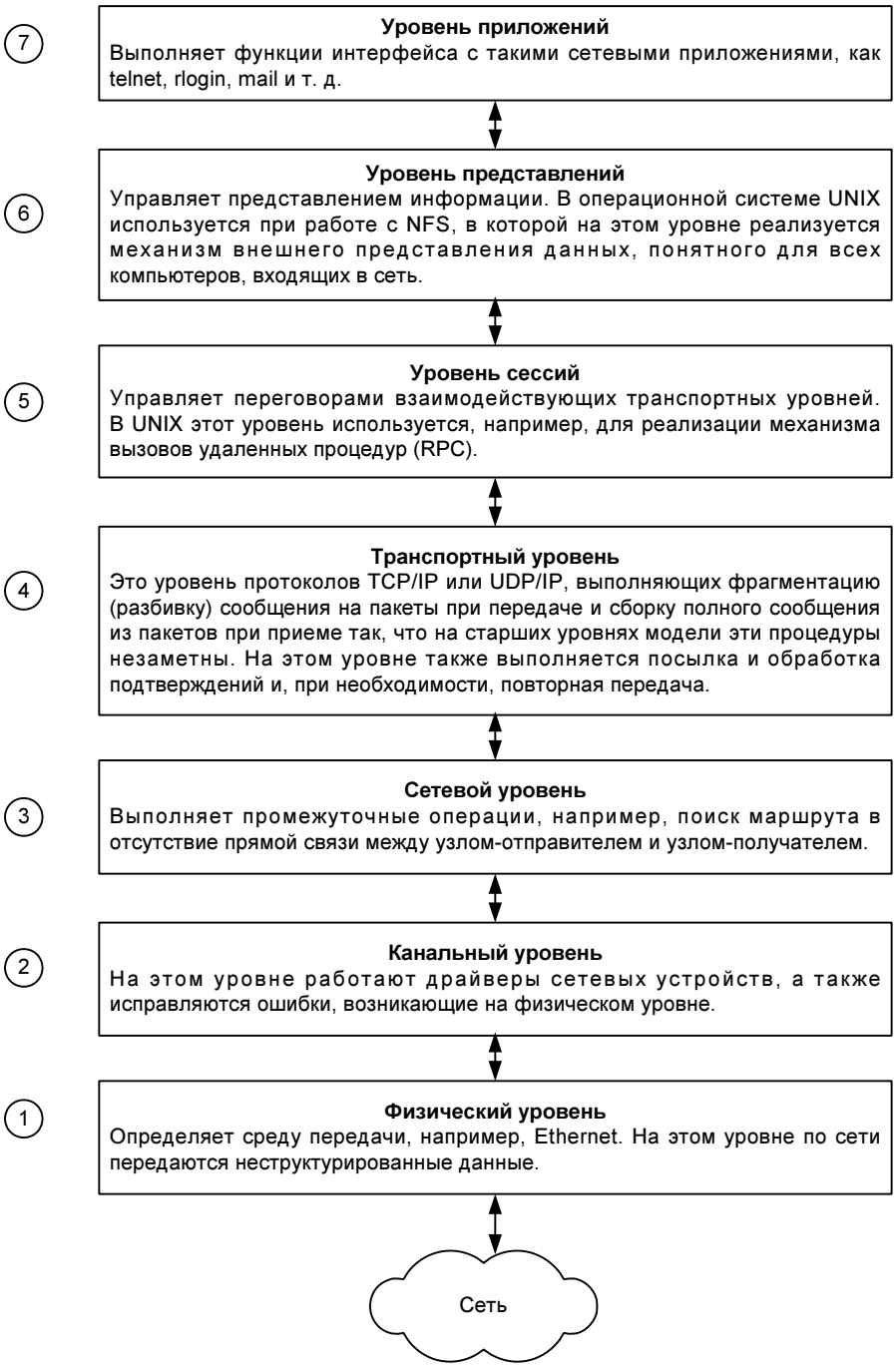


Рис. 8.1. Семиуровневая модель OSI стека протоколов TCP/IP

- **Канальный уровень.** Здесь обеспечивается передача кадра данных между любыми узлами в сетях с типовой топологией либо между двумя соседними узлами в сетях с произвольной топологией. В протоколах канального уровня заложена определенная структура связей между компьютерами и способы их адресации. Адреса, используемые на канальном уровне в локальных сетях, часто называют аппаратными или MAC-адресами. Протоколы канального уровня не позволяют строить сети с развитой структурой, например, сети, объединяющие несколько сетей предприятия в единую сеть, или высоконадежные сети с избыточными связями между узлами.
- **Сетевой уровень.** Этот уровень отвечает за доставку данных между любыми двумя узлами в сети с произвольной топологией, не гарантируя при этом надежности передачи данных. Сетевой уровень занимает в модели OSI промежуточное положение — к его услугам обращаются протоколы прикладного уровня, сеансового уровня и уровня представления. Смысл использования сетевого уровня состоит в том, чтобы, не затрагивая технологий, используемых в объединяемых сетях, добавить в кадры канального уровня дополнительную информацию в виде заголовка пакета сетевого уровня. Такой заголовок позволяет находить адресата в сети с любой базовой технологией, причем он имеет унифицированный формат, не зависящий от форматов кадров канального уровня сетей, входящих в объединенную сеть. Обычно заголовок сетевого уровня содержит адрес назначения и другую информацию, необходимую для успешного перехода пакета из сети одного типа в сеть другого типа. На этом уровне функционируют протоколы, отвечающие за отображение адреса узла в локальный адрес сети. Такие протоколы называют протоколами разрешения адресов — Address Resolution Protocol (ARP). Иногда их относят к канальному уровню, хотя тонкости классификации не изменяют их сути.
- **Транспортный уровень.** На этом уровне обеспечивается передача данных между любыми узлами сети с требуемым уровнем надежности, поэтому транспортный уровень предполагает наличие средств для установления соединения, а также нумерации, буферизации и упорядочивания пакетов.
- **Сеансовый уровень.** Функции этого уровня предоставляют средства управления диалогом между взаимодействующими сторонами, позволяющие синхронизировать операции в рамках процедуры обмена сообщениями.
- **Уровень представления.** В отличие от более низких, этот уровень обеспечивает внешнее представление данных. Здесь могут выполняться различные виды преобразования данных, такие как компрессия и декомпрессия, шифровка и дешифровка данных.

- **Прикладной уровень.** На этом уровне работают различные сетевые сервисы, предоставляющие услуги конечным пользователям и приложениям, например, электронная почта, передача файлов, подключение удаленных терминалов к компьютеру по сети.

Следует сказать, что семиуровневая модель OSI является скорее теоретической абстракцией, дающей хорошее представление о сетевом взаимодействии на разных уровнях, но не реализована в практическом плане для большинства приложений. На практике разработчики программного обеспечения обходятся четырьмя уровнями, как показано на рис. 8.2.

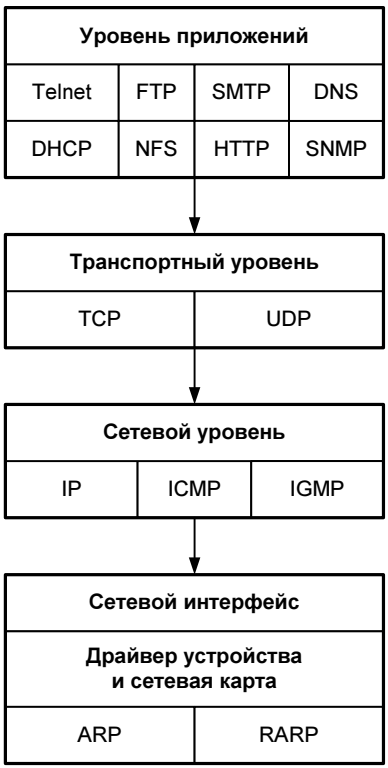


Рис. 8.2. Практическая 4-уровневая модель сетевого взаимодействия

Все четыре уровня в этой практической модели выполняют определенные функции, включая в себя те или иные протоколы. При этом некоторые уровни в этой модели объединяют функциональность нескольких уровней в классической семиуровневой модели OSI.

Функции каждого из четырех уровней описаны далее.

- **Сетевой интерфейс.** Этот уровень отвечает за установление сетевого соединения в конкретной физической сети, к которой подсоединен компьютер. На этом уровне работают драйвер устройства в операционной системе и соответствующая сетевая плата компьютера. На самом нижнем уровне сетевого интерфейса используются специальные протоколы разрешения адресов ARP (Adress Resolution Protocol) и RARP (Reverse Adress Resolution Protocol). Они применяются только в определенных типах физических сетей (Ethernet и Token Ring) для преобразования адресов сетевого уровня в адреса физической сети и обратно.
- **Сетевой уровень.** На этом уровне реализуется ненадежная служба доставки пакетов по сети от системы к системе без установления соединения, в частности, маршрутизация пакетов по Интернету. Это означает, что будут выполнены все необходимые операции для доставки пакетов, хотя сама доставка не гарантируется: пакеты могут быть потеряны, переданы в неправильном порядке, продублированы и т. д. На этом уровне принимается решение о маршрутизации пакета по межсетевым соединениям, в которых используется протокол IP — основной протокол сетевого уровня. Он используется протоколами TCP и UDP, функционирующими на транспортном уровне. Протокол IP определяет базовую единицу передачи данных в Интернете — IP-дейтаграмму, указывая точный формат всей информации, проходящей по сети TCP/IP. Программное обеспечение IP выполняет функции маршрутизации при выборе пути данных по физическим сетям. Для выполнения таких функций поддерживаются специальные таблицы, а выбор осуществляется на основе адреса сети, в которой находится компьютер-адресат. Протокол IP определяет маршрут отдельно для каждого пакета данных, не гарантируя надежной доставки в нужном порядке. Он задает непосредственное отображение данных на нижележащий физический уровень передачи, реализуя тем самым доставку пакетов. Кроме протокола IP, на сетевом уровне используются также протоколы ICMP (Internet Control Message Protocol) и IGMP (Internet Group Management Protocol). Протокол ICMP отвечает за обмен сообщениями об ошибках и другой важной информацией с сетевым уровнем на другом хосте или маршрутизаторе, а IGMP используется для отправки IP-дейтаграмм множеству хостов в сети.
- **Транспортный уровень.** Этот уровень реализует надежную передачу данных, при этом работающие на данном уровне два основных протокола, TCP и UDP, осуществляют связь между машиной — отправителем пакетов и машиной-адресатом. При анализе протоколов транспортного уровня часто возникает вопрос: зачем нужны два транспортных протокола TCP и UDP?

Дело в том, что каждый из этих протоколов предоставляет определенные услуги прикладным процессам. Как правило, большинство прикладных программ использует только один из них. TCP (Transmission Control Protocol) обеспечивает надежную передачу данных между двумя хостами. Он позволяет клиенту и серверу приложения устанавливать между собой логическое соединение и затем использовать его для передачи больших массивов данных, как если бы между ними существовало прямое физическое соединение. Протокол позволяет осуществлять дробление потока данных, подтверждать получение пакетов данных, задавать тайм-ауты (которые разрешают подтвердить получение информации), организовывать повторную передачу в случае потери данных и т. д. Поскольку данный транспортный протокол реализует гарантированную доставку информации, то использующие его приложения получают возможность игнорировать все детали такой передачи. Протокол UDP (User Datagram Protocol) реализует гораздо более простой сервис передачи, обеспечивая, подобно протоколам сетевого уровня, ненадежную доставку данных без установления логического соединения, но, в отличие от IP, для прикладных систем на хост-компьютерах. Он просто посылает пакеты данных, дейтаграммы (datagrams) с одной машины на другую, но не предоставляет никаких гарантий их доставки. Все функции надежной передачи должны встраиваться в прикладную систему, использующую UDP. Протокол UDP имеет и некоторые преимущества перед TCP. Для установления логических соединений требуется определенное время и дополнительные системные ресурсы для поддержки на компьютере информации о состоянии соединения. UDP занимает системные ресурсы только в момент отправки или получения данных. Поэтому если распределенная система осуществляет непрерывный обмен данными между клиентом и сервером, связь с помощью транспортного уровня TCP окажется для нее более эффективной. Если же коммуникации между хостами осуществляются редко, предпочтительней использовать протокол UDP. Среди известных приложений, использующих TCP, такие как telnet, FTP и SMTP. Протоколом UDP пользуется, в частности, протокол сетевого управления SNMP.

- **Уровень приложений.** На этом уровне выполняются приложения типа "клиент-сервер", базирующиеся на протоколах нижних уровней. В отличие от протоколов остальных трех уровней протоколы этого уровня работают непосредственно с приложением, не вдаваясь в детали передачи данных по сети. К протоколам, работающим на этом уровне, можно отнести telnet, протокол передачи файлов FTP, протокол электронной почты SMTP, протокол управления сетью SNMP, используемый в World Wide Web, протокол передачи гипертекста HTTP и др.

Рассмотрим более подробно использование протоколов уровня приложений, с которыми имеет дело подавляющее большинство пользователей. Сетевые приложения могут работать либо как серверы, обслуживая запросы приложений-клиентов, либо как клиенты, запрашивающие приложения-серверы для получения необходимой им информации или других действий. И клиенты, и серверы могут использовать непосредственно как протоколы TCP/IP, так и протоколы более высокого уровня, например, уровня приложений (Telnet, FTP и т. д.).

Здесь следует уточнить некоторые термины и понятия, которые будут в дальнейшем очень часто использоваться при анализе сетевых настроек UNIX. Так, под термином "сетевой сервис" или, что то же самое, "сетевая служба" будем понимать программный интерфейс, реализующий тот или иной протокол и выполняющийся как процесс-сервер. Процесс-сервер, реализующий сетевой сервис, может использовать один или несколько файлов конфигурации, баз данных, а также запускать другие вспомогательные программы. Процесс-сервер обычно функционирует как демон операционной системы UNIX, хотя может выполняться и как обычное приложение. Например, сетевой сервис telnet обычно реализуется как процесс-демон `telnetd`. Демон `telnetd` очень часто называют сервером telnet. Данная терминология применима ко всем сетевым сервисам UNIX. Так, например, демон `ftpd`, реализующий сервис FTP, принято называть сервером FTP.

Хочу сделать очень важное замечание. Клиентское приложение, выполняющее обмен данными по какому-либо протоколу, не может реализовать сетевой сервис по определению. Тем не менее, можно создать приложение, которое может выполнять как функцию сервера службы, так и при определенных условиях осуществлять клиентские функции.

Протоколы TCP и UDP идентифицируют приложения по 16-битовым номерам портов. Серверы сетевых приложений обычно имеют заранее известные номера портов, которые используются программами-клиентами для запросов на обслуживание. Например, сервер FTP получает номер TCP-порта 21, Telnet-сервер имеет TCP-порт 23, а сервер протокола TFTP (Trivial File Transfer Protocol) — UDP-порт 69. Сервисам, которые поддерживаются в любой реализации TCP/IP, назначаются номера портов в диапазоне от 1 до 1023. Назначение номеров портов находится в ведении организации IANA (Internet Assigned Numbers Authority).

Перейдем к обсуждению наиболее широко используемых протоколов уровня приложений и начнем с telnet.

Протокол telnet реализован на базе протокола TCP и позволяет работать с удаленным компьютером через стандартный виртуальный терминал строчного

типа в кодировке ASCII, обеспечивая при этом выполнение довольно сложных функций, таких как локальный или удаленный эхоконтроль, страничный режим и т. д. При этом на серверной стороне должен работать процесс-демон `telnetd` (`in.telnetd` в операционных системах Solaris), к которому обращается программа-клиент.

Демон `telnetd` обычно запускается посредством суперсервера `inetd` (хотя может быть запущен и вручную) и прослушивает TCP-порт 23 в ожидании соединений. При поступлении запроса от клиента на обслуживание, `telnetd` открывает для каждого удаленного клиента виртуальный терминал (псевдотерминал), с которым связывает стандартный ввод/вывод, а также стандартное устройство ошибок.

В процессе взаимодействия с клиентом `telnetd` посылает ему команды настройки (эхо, обмен двоичной информацией, тип терминала, скорость обмена, переменные окружения). Обмен данными происходит с использованием стандартных команд протокола telnet.

Для вызова сервиса telnet клиент (пользователь) может воспользоваться либо командной строкой, либо установить связь в режиме удаленного терминала. Работа в режиме удаленного терминала позволяет использовать как буферизованный, так и посимвольный ввод/вывод. При работе в посимвольном режиме каждый введенный символ немедленно отправляется на удаленную машину, откуда приходит эхо-ответ. Если используется буферизованный ввод/вывод, то введенные символы накапливаются в буфере памяти клиентской программы и отправляются на удаленную машину в виде пакета данных.

В практическом плане для начала сеанса telnet пользователь вводит с консоли командную строку наподобие

```
telnet host
```

Здесь *`host`* — имя удаленного компьютера (хоста) или его IP-адрес. Если соединение было успешным, пользователь получает на экране приглашение ко входу в машину *`host`*.

Протокол FTP (File Transfer Protocol, протокол передачи файлов), так же как и telnet, распространен очень широко и в качестве транспортного протокола использует TCP. С помощью FTP можно просмотреть, например, каталог удаленной машины, перейти из одного каталога в другой, а также скопировать один или несколько файлов. Архивы FTP представляют собой один из основных информационных ресурсов Интернета и содержат огромное количество различной информации, включая текстовые документы, программное обеспечение, аудио- и видеоинформацию, хранящуюся в виде файлов на компьютерах во всем мире. Компьютеры, взаимодействующие по FTP-

протоколу, могут находиться как в одной, так и в разных сетях, причем с разными установленными операционными системами (не только UNIX), поскольку протокол FTP поддерживается многими системами.

Как и многие другие виды сервиса, FTP осуществляет обмен данными по принципу "клиент-сервер". Программа-сервер реализована в виде демона `ftpd` (`in.ftpd` в Solaris), который принимает входящие запросы на TCP-порт 21. Демон `ftpd` запускается суперсервером `inetd`, хотя может выполняться и как изолированное приложение-сервер.

Пользовательский интерфейс, предоставляемый протоколом FTP, работает по принципу интерпретатора команд на удаленной машине. Пользователь может вводить UNIX-подобные команды для выполнения тех или иных действий на удаленной машине.

В настоящее время большинство операционных систем и коммуникационных приложений включает те или иные реализации FTP-протокола, которые могут незначительно отличаться друг от друга набором команд. В табл. 8.1 приводится список команд, которыми оперируют программы FTP большинства UNIX-систем. В любом случае перед использованием FTP желательно просмотреть документацию для конкретной системы, чтобы определить команды, совпадающие с имеющимися в таблице.

Таблица 8.1. *Команды FTP*

Команда	Выполняемая функция
`?`	Отображает справочную информацию о FTP-командах
`ascii`	Устанавливает режим передачи файла в ASCII (каждый символ передается последовательностью в 7 битов). Данный режим устанавливается по умолчанию
`binary`	Устанавливает двоичный режим передачи файлов, при котором передаются все 8 битов одного байта. Это обеспечивает более надежную передачу информации по сравнению с ASCII-режимом. Используя данный режим, можно передавать файлы любого формата, а не только текстовые в формате ASCII
`bye`	По этой команде происходит выход из сеанса FTP (по аналогии с командой `quit`)
`cd`	Позволяет перейти в другой каталог на удаленной машине
`close`	Позволяет закрыть соединение с другим компьютером
`delete`	Позволяет удалить файл из текущего каталога на удаленной машине (то же самое, что и команда `rm` на локальной UNIX-системе)
`get`	Позволяет копировать один файл из удаленного хоста на локальную машину

Таблица 8.1 *(окончание)*

Команда	Выполняемая функция
`help`	Отображает список всех FTP-команд
`lcd`	Позволяет сменить каталог на локальной машине (то же самое, что и команда `cd` локальной UNIX-системы)
`ls`	Отображает список файлов из текущего каталога на удаленной машине
`mkdir`	Позволяет создать новый каталог в текущем каталоге на удаленной машине
`mget`	Позволяет копировать несколько файлов с удаленной на локальную машину, при этом требуется ввести подтверждение операции перед копированием каждого файла
`mput`	Позволяет копировать несколько файлов с локальной на удаленную машину, при этом требуется ввести подтверждение операции перед копированием каждого файла
`open`	Устанавливает соединение с другим хостом
`put`	Позволяет скопировать один файл из локальной машины на удаленную
`pwd`	Отображает путевое имя текущего каталога на удаленной машине
`quit`	По этой команде происходит выход из сеанса FTP (по аналогии с командой `bye`)
`rmdir`	Удаляет каталог в текущем каталоге на удаленной машине

Вот несколько примеров использования команд FTP. Команда

```
close remote_host
```

закрывает текущее FTP-соединение с хостом `remote_host`, при этом не происходит выход из программы `ftp`.

Команда

```
get remote_file local_file
```

копирует файл remote_file из текущего удаленного каталога в файл local_file в текущий каталог на локальной машине.

Команда

```
get remote_file
```

копирует файл remote_file из текущего удаленного каталога в файл с таким же именем в текущий каталог на локальной машине.

Команда

```
mget *
```

копирует все файлы из текущего удаленного каталога в файлы с теми же именами в текущий каталог на локальной машине.

Команда

```
open remote_host
```

открывает новое FTP-соединение с удаленной машиной `remote_host`. При этом потребуется ввести имя пользователя и пароль, если только это не соединение для анонимного пользователя (anonymous FTP-connection).

Чтобы открыть FTP-соединение с удаленным узлом, нужно в командной строке ввести команду

```
ftp имя_узла
```

Здесь *имя_узла* — полное имя удаленного узла, например, **ftp.freebsd.org** и т. д. Следует учитывать, что на данном узле должен функционировать FTP-сервер, иначе провести сеанс не удастся. Если имя узла неизвестно, можно использовать его IP-адрес, например,

```
ftp 129.82.45.181
```

При успешной установке соединения FTP-сервер потребует ввода имени и пароля. Если введенные имя и пароль являются корректными, то на консоли отобразится приглашение

```
ftp>
```

а пользователь получит доступ к своему домашнему каталогу на удаленной машине, откуда он может считывать и куда может записывать файлы с помощью команд FTP.

Для большей безопасности доступ к FTP-серверу можно установить для определенных пользователей, потребовав при открытии сеанса пароль. Тем не менее, одним из самых распространенных видов FTP-серверов является так называемый анонимный FTP-сервер (anonymous FTP), позволяющий пользователю копировать файлы с удаленного хоста, не имея там учетной записи, а используя анонимный доступ. Для этого при запросе имени пользователя следует ввести слово anonymous, а при запросе пароля — свой адрес электронной почты (это позволяет FTP-серверу вести статистику запросов). Во многих случаях пароль вообще не требуется — нужно нажать клавишу <Enter>.

После входа на сервер анонимный пользователь обычно попадает в специально выделенный домашний каталог, где находятся ресурсы, доступные для копирования на свою локальную машину. Тем не менее, удалить файлы из

каталога анонимного пользователя или записать их туда нельзя из-за отсутствия доступа на выполнение этих операций.

Далее приводится пример сеанса FTP, в котором анонимный пользователь копирует файл README с узла **ftp.kde.org** (команды выделены жирным шрифтом):

```
[root@yuryhost root]# ftp ftp.kde.org
Connected to ftp.kde.org (131.220.60.97).
220 welcome to the bolug ftp server! -- contact: info@bolug.uni-bonn.de
Name (ftp.kde.org:root): anonymous
331 Please specify the password.
Password:
230 Login successful. Have fun.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
227 Entering Passive Mode (131,220,60,97,238,190)
150 Here comes the directory listing.
drwxr-xr-x    2 root     root         4096 Apr 27  2001 bin
drwxr-xr-x    2 root     root         4096 Apr 27  2001 dev
drwxr-xr-x    2 root     root         4096 Oct 21  2002 etc
drwxr-xr-x    2 root     root         4096 Apr 27  2001 lib
drwxr-xr-x    2 root     root         4096 May 02  2002 msgs
drwxr-xr-x    8 root     root         4096 Jul 09  2004 pub
drwxr-xr-x    3 root     root         4096 Apr 27  2001 usr
226 Directory send OK.
ftp> cd pub
250 Directory successfully changed.
ftp> ls
227 Entering Passive Mode (131,220,60,97,136,19)
150 Here comes the directory listing.
drwxr-xr-x    7 ftpuser  ftpgroup     4096 Dec 09  2002 bolug
drwxrwxr-x   13 ftpuser  ftpgroup     4096 Apr 04  2004 kde
drwxr-xr-x    2 root     root         4096 Dec 09  2002 konq-e
drwxr-xr-x    2 mipsadm  ftpgroup     4096 Apr 26  2003 mips-linux
drwxr-xr-x    2 mipsadm  ftpgroup     4096 Apr 26  2003 mipsel-linux
drwxr-xr-x    7 ftpuser  ftpgroup     4096 Apr 16 15:03 suse
226 Directory send OK.
ftp> cd kde
```

```
250-
250-              This is the ftp distribution of
250-
250-                The KDE Project
250-                http://www.kde.org/
250-
250-
250----------------------------------------------------------------------
250-
250-Please note:
250-
250- The latest stable version of the KDE base distribution
250- can be found in the 'stable' subdirectory.
250-
250-- You are welcome to upload contributions to
250-
250- ftp://upload.kde.org/incoming
250-
250-Your Friendly ftp.kde.org Archive Main
250 Directory successfully changed.
ftp> ls
227 Entering Passive Mode (131,220,60,97,203,206)
150 Here comes the directory listing.
drwxrwxr-x   20 ftpuser  ftpgroup     4096 Apr 19 08:54 Attic
-rw-rw-r--    1 ftpuser  ftpgroup     1027 Oct 24  2002 README
-rw-rw-r--    1 ftpuser  ftpgroup      404 Apr 04  2004 README_UPLOAD
drwxrwxr-x    3 ftpuser  ftpgroup     4096 Oct 01  2000 adm
drwxrwxr-x    3 ftpuser  ftpgroup     4096 Mar 07  2005 contrib
drwxrwxr-x   11 ftpuser  ftpgroup     4096 Jan 08  2004 devel
drwxrwxr-x    5 ftpuser  ftpgroup     4096 Mar 17  2002 doc
drwxrwxr-x    5 ftpuser  ftpgroup     4096 Dec 07  2000 events
-rw-rw-r--    1 ftpuser  ftpgroup  1953670 May 01 02:03 ls-lR
-rw-rw-r--    1 ftpuser  ftpgroup   173944 May 01 10:03 ls-lR.bz2
drwxrwxr-x    4 ftpuser  ftpgroup     4096 Apr 21  2004 packages
drwxrwxr-x    2 ftpuser  ftpgroup     4096 Apr 07 07:55 printing
drwxrwxr-x    2 ftpuser  ftpgroup    12288 Apr 04 10:05 security_patches
lrwxrwxrwx    1 ftpuser  ftpgroup       18 Sep 11  2002 snapshots -> un-
stable/snapshots
drwxr-xr-x   14 ftpuser  ftpgroup     4096 Apr 19 08:55 stable
```

```
drwxr-xr-x    8 ftpuser  ftpgroup     4096 Mar 28 09:19 unstable
226 Directory send OK.
ftp> lcd
Local directory now /root
ftp> get README
local: README remote: README
227 Entering Passive Mode (131,220,60,97,193,28)
150 Opening BINARY mode data connection for README (1027 bytes).
226 File send OK.
1027 bytes received in 0.22 secs (4.6 Kbytes/sec)
ftp> bye
221 Goodbye.
[root@yuryhost root]#
```

Как видно из этого листинга, для входа на FTP-сервер пароль не требуется (другие серверы могут потребовать ввода адреса электронной почты в качестве пароля, выдавая при этом соответствующее приглашение). Кроме того, сервер сразу же устанавливает двоичный режим, предпочтительный для передачи файлов (большинство FTP-серверов использует данный режим по умолчанию, хотя в отдельных случаях его можно установить вручную командой `binary`). Смысл команд, вводимых пользователем, думаю, понятен и в комментариях не нуждается.

Протокол SMTP (Simple Mail Transfer Protocol, простой протокол передачи почты) обеспечивает передачу сообщений электронной почты между произвольными узлами сети Интернет. Имея механизмы промежуточного хранения почты и механизмы повышения надежности доставки, протокол SMTP допускает использование различных транспортных служб. При этом он может функционировать даже в тех сетях, которые не используют протоколы TCP/IP. Протокол SMTP обеспечивает группирование сообщений в адрес одного получателя, а также размножение нескольких копий сообщения для передачи в разные адреса. Этот протокол будет рассмотрен более подробно при анализе работы электронной почты.

Еще один протокол SNMP (Simple Network Management Protocol, простой протокол управления сетью) работает на базе UDP и применяется для контроля сетевых управляющих станций. Рабочие станции, использующие этот протокол, могут собирать информацию о состоянии сети в определенном формате, после чего можно обрабатывать и интерпретировать полученные данные.

Протокол TCP лежит в основе функционирования системы X Window, позволяющей работать с графикой и текстом и являющейся основой графического интерфейса пользователя в подавляющем большинстве операционных систем.

Сетевая файловая система NFS (Network File System) использует транспортные услуги протокола UDP, позволяя монтировать файловые системы нескольких UNIX-машин и работать с ними как с локальными ресурсами. К более подробному анализу NFS мы вернемся немного позже.

Работу многочисленных хостов в Интернете обеспечивает еще один протокол — HTTP, которому мы уделим достаточно внимания в последующих главах, и который также базируется на стеке протоколов TCP/IP.

Кроме упомянутых сетевых сервисов чрезвычайно важным является так называемый сервис имен или служба разрешения имен (DNS), лежащая в основе работы как локальных сетей, так и Интернета. Принципы функционирования данной службы мы рассмотрим подробно далее в этой главе. Наконец, протокол динамического присвоения адресов (DHCP), который позволяет автоматизировать сетевые настройки компьютеров, также будет проанализирован далее.

Хотелось бы остановиться на некоторых общих вопросах, касающихся функционирования сетевых служб. Имена большинства сетевых сервисов и порты, на которых они работают, описаны в файле /etc/services. Так, для telnet и FTP полученная информация может выглядеть так:

```
# cat /etc/services|grep telnet;cat /etc/services|grep ftp
telnet          23/tcp
ftp-data        20/tcp
ftp             21/tcp
tftp            69/udp
```

Из результата видно, что на этом хосте служба telnet для обмена данными использует TCP-порт 23, а служба FTP — TCP-порт 21.

Подробную информацию о работающих сетевых приложениях можно получить и при помощи утилиты `netstat` (ее мы рассмотрим далее в этой главе), которая реализована во всех версиях операционной системы UNIX.

Ранее уже упоминалось, что сетевые серверы (telnet, FTP, SMTP, DNS, NFS, SNMP и т. д.) могут оперировать одновременно со многими клиентами. Рассмотрим, как это происходит на практике.

Клиентское приложение обычно использует уникальный номер порта, назначенный ему сервером при подключении, и работает с этим портом в течение сеанса. Это позволяет процессам-серверам обслуживать несколько клиентских приложений одновременно. Номера портов клиентских приложений называют краткосрочными, поскольку клиенты существуют ровно столько времени, сколько работающий с ним пользователь нуждается в соответствующем сервере. Серверы, напротив, находятся в рабочем состоянии все время, пока включен

хост, на котором они работают. В большинстве реализаций TCP/IP краткосрочным номерам портов выделен диапазон от 1024 до 5000.

Вышеизложенное можно проиллюстрировать на рис. 8.3, где показано взаимодействие сервера telnet с клиентами.

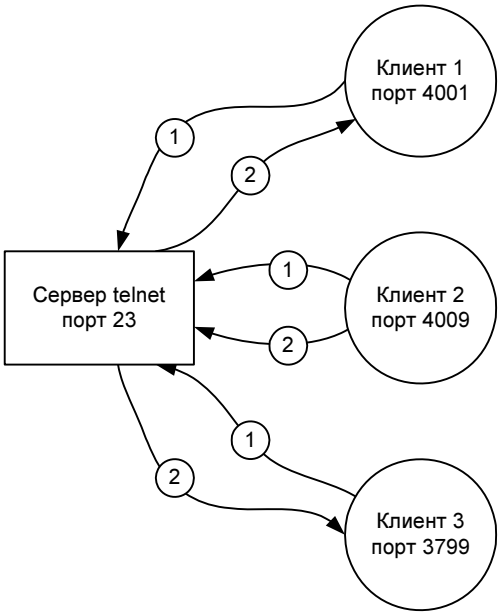


Рис. 8.3. Пример сетевого взаимодействия типа "клиент-сервер"

Из этого рисунка видно, что клиент запрашивает соединение по стандартному порту 23 (1). Сервер принимает соединение на этом порту и создает новое (рабочее) соединение для каждого из клиентов (2). При этом для клиента 1 обмен данными осуществляется по порту 4001, для клиента 2 — по порту 4009 и для клиента 3 — по порту 3799. Закончив сеансы обмена данными, сервер должен закрыть соединения для клиентов. В свою очередь, каждый из клиентов должен закрыть инициированное им соединение.

Хочу обратить ваше внимание на то, что, работая с данными клиентами, сервер telnet продолжает прослушивать порт 23 в ожидании новых соединений. Подобную схему использует подавляющее большинство сетевых приложений TCP, таких как telnet, FTP, SMTP и т. д.

Следует сказать, что помимо упомянутых протоколов уровня приложения существует целый ряд других, которые не столь часто применяются пользователями, например, протокол удаленного доступа rlogin, протокол удален-

ного выполнения команд rsh и т. д. Эти протоколы здесь рассматриваться не будут, а читатели без труда смогут найти необходимую информацию в Интернете или в документации к операционным системам.

8.1. Адресация в Интернете

Компьютеры, взаимодействующие в сети, должны обладать методом идентификации, позволяющим однозначно определить, каким образом можно получить доступ к каждому из них. В TCP/IP выбрана схема идентификации, при которой каждому сетевому интерфейсу (сетевой карте либо последовательному порту) присваивается уникальный 32-битовый адрес (IP-адрес), используемый для всех коммуникаций с этим интерфейсом по Интернету. IP-адрес компьютера имеет определенную структуру, позволяя задавать идентификатор сети, к которой подсоединен компьютер, а также уникальный идентификатор самого компьютера.

IP-адрес — это четырехбайтовое число, записываемое либо в шестнадцатеричном виде типа 0xC0A80E05, либо в десятичном виде, где байты разделены точками, например, 192.168.14.5 (в качестве примера в обоих случаях использовался один и тот же номер).

Вот еще примеры записи IP-адресов в десятичной нотации:

- 193.124.148.73
- 128.8.2.1

Еще одним обязательным атрибутом является маска сети, которая определяет размер сети, т. е. задает число адресов в сети. Маска представляет собой четырехбайтовое число, при этом все старшие биты, начиная с некоторого, всегда установлены в единицу, а все младшие — в ноль. Маска может быть задана либо количеством битов (например, 8 битов — 256 адресов, 6 битов — 64 адреса), либо в виде последовательности битов:

```
b'111...11100...00'
```

Маску можно записывать и в десятичной нотации, например:

- 255.255.255.192 — маска на 64 адреса;
- 255.255.255.0 — маска на 256 адресов;
- 255.255.0.0 — маска на 64 Кбайт адресов.

Вот еще примеры масок сети:

- 255.255.255.0 — маска сети класса C на 256 номеров;
- 255.255.255.192 — маска небольшой сети на 64 адреса (192 = 256 – 64).

В отдельной сети все машины (хосты) имеют IP-адреса с одинаковым адресом сети и одинаковой маской. В одной локальной сети можно совместить две и больше разных IP-сетей, они даже могут не знать о существовании друг друга и, тем не менее, обмениваться данными.

Самый старший адрес в сети используется для широковещательной (broadcast) рассылки, например, 128.8.255.255.

Если нужно указать сочетание номера и маски, то можно использовать запись "номер/число_установленных_битов_в_маске". Так, сочетание IP-адреса 192.168.14.5 и маски 255.255.255.0 будет записано в виде 192.168.14.5/24.

Сами же сети делятся на такие классы:

- класс A —"большие" сети. Адреса этих сетей лежат в диапазоне 1—126, маска сети равна 255.0.0.0, а сама сеть содержит до 16 777 216 адресов (256×256×256). Адреса хостов в этих сетях имеют вид "125.*.*.* ";
- класс B —"средние" сети. Адреса этих сетей лежат в диапазоне 128.0—191.255, маска сети равна 255.255.0.0, а сама сеть содержит до 65 536 адресов (256×256). Адреса хостов в этих сетях имеют вид "136.12.*.* ";
- класс C — "небольшие" сети. Адреса сетей лежат в интервале 192.0.0—255.254.255, маска сети равна 255.255.255.0, а сама сеть содержит 254 адреса. Адреса хостов в этих сетях имеют вид "195.136.12.* ".

Многие приложения по адресу автоматически определяют класс сети, хотя это можно поправить вручную. В принципе ничто не мешает разбить сеть на две или больше подсетей с любыми масками, хотя организациям, как правило, выделяют адреса блоками, соответствующими классам A, B и C, что связано с системой доменных имен (DNS).

Сеть класса A с номером 127 — это специальный вид сети, известной под названием "интерфейс обратной связи" (loopback interface), которая предназначена для настройки и проверки сетевого интерфейса самого компьютера.

Сети класса C используются в небольших организациях, объединяющих до 255 машин, их адреса лежат в диапазоне 192.0.0.0—223.255.255.255, при этом 21 бит отводится для идентификатора сети и 8 битов — для идентификатора хоста. В более крупных организациях используются сети класса B, способные обслужить до 256 сетей, причем каждая из них может содержать до 64 тыс. рабочих станций. Наконец, сети класса A используются в глобальных сетях очень большого масштаба, таких, например, как ARPANET.

Адресом (номером) сети называют число, получаемое из номера интерфейса применением побитовой операции AND с маской, т. е. в номере интерфейса обнуляются биты на тех местах, на которых стоят нулевые биты в маске. Каждая сеть имеет свой уникальный номер, например, 194.15.28.0, который

не может быть присвоен никому конкретно. Последний номер в сети предназначен для передачи широковещательных сообщений (broadcasting), которые доставляются всем машинам данного сегмента сети, например, 194.15.28.255. Таким образом, при выделении группы адресов в сеть два адреса (сети и широковещательной рассылки) становятся недоступными для конкретных хостов.

Компьютеры могут находиться не в одной, а в нескольких физических сетях, при этом им присваивается несколько IP-адресов — по одному для каждой сети. Помимо индивидуальных адресов, в сети выделяют ряд специальных адресов, куда входят широковещательные (broadcast) и групповые (multicast) адреса.

С помощью широковещательных адресов можно обращаться ко всем хостам сети, при этом такие адреса имеют специальный идентификатор, состоящий только из единиц. Механизм широковещательной передачи существенно зависит от характеристик конкретной физической сети. В сети Ethernet, например, широковещательная передача может выполняться так же, как и обычная передача данных, хотя есть сети, которые вообще не поддерживают такой тип передачи или имеют для этого ограниченные возможности.

Групповые адреса (их иногда называют адресами класса D) позволяют отправлять сообщения определенному множеству адресатов (multicasting). Многие приложения используют групповую адресацию, например, программы интерактивных конференций, отправки почты или новостей группе получателей. Для поддержки групповой передачи хосты и маршрутизаторы используют протокол IGMP, предоставляющий всем системам в физической сети информацию о принадлежности хоста к той или иной группе.

Каждая сетевая карта или, в терминологии UNIX, каждый сетевой интерфейс имеет уникальный 32-битовый адрес. Тем не менее, в связи с бурным ростом Интернета 32-битовая схема адресации нынешней версии IPv4 протокола IP уже не удовлетворяет потребностей Всемирной сети. В связи с этим была разработана новая версия протокола IP, известная как IPv6, проект которой был обнародован в 1991 году. Стандарт IPv6 предполагает использование 128-битового формат IP-адреса, и, кроме того, поддерживает автоматическое назначение адресов.

В данной главе мы ограничимся рассмотрением особенностей сетевых настроек UNIX только для версии IPv4.

Установка IP-адреса для одной машины не является сложной задачей, но для корпоративной сети, содержащей большое число компьютеров, которые могут постоянно перемещаться, управление IP-адресами вручную представляет собой довольно сложную задачу. Если, например, компьютер перемещается

в другую подсеть, то следует изменить его сетевую идентификацию таким образом, чтобы часть адреса с номером хоста не конфликтовала с адресами узлов новой подсети. Для автоматизации задач управления сетями существуют определенные решения, способствующие повышению эффективности использования адресов, а также средства организации виртуальных частных сетей.

Для автоматизации управления сетями был разработан протокол динамической конфигурации хоста, известный под названием DHCP (Dynamic Host Configuration Protocol). В соответствии с этим протоколом одна из машин сети настраивается как сервер DHCP, а все остальные становятся его клиентами. Протокол DHCP поддерживает динамическое назначение IP-адресов, позволяя присвоить хосту IP-адрес из заданного при конфигурации сервера диапазона IP-адресов. При этом адрес выделяется на ограниченный период времени или до отказа клиента от его использования, что позволяет эффективно использовать ограниченное число IP-адресов.

Протокол DHCP, кроме поддержки автоматического присвоения постоянного IP-адреса, позволяет назначать адреса вручную, при этом сервер DHCP только извещает клиента о назначении. Реализация клиентской части протокола поддерживается большинством операционных систем; более того, TCP/IP дает пользователям возможность работать не с адресами хостов, а с их символьным представлением (именами), что намного удобнее для восприятия. Реализация такого механизма отображения IP-адресов на имена хостов реализована в виде распределенной базы данных DNS (Domain Name System). При этом любая программа может вызвать стандартную библиотечную функцию для преобразования IP-адреса в соответствующее имя хоста или наоборот.

База данных DNS является распределенной и включает информацию только об ограниченном числе хостов в пределах определенного региона. Многие организации и компании поддерживают свою базу данных DNS, причем к ней могут обращаться другие клиенты сети. Кроме того, распределенная база DNS, содержащаяся на сервере, взаимодействует с клиентами посредством определенного протокола.

8.2. Сетевые интерфейсы

Одной из самых первых задач при настройке сети в UNIX является настройка сетевого интерфейса. Во всех реализациях операционной системы для решения этой задачи используется утилита `ifconfig`. Утилита имеет множество опций, отличающихся в различных операционных системах, и применяется

на этапе загрузки для настройки интерфейсов или в процессе отладки и диагностики в случае необходимости. Программа имеет синтаксис:

```
ifconfig [интерфейс]
ifconfig интерфейс [семейство_адресов] опции | адрес ...
```

Заданная без аргументов, `ifconfig` выдает информацию о состоянии активных интерфейсов. Если указан один аргумент *интерфейс*, выдается информация только о состоянии этого интерфейса; если указан один аргумент `-a`, выдается информация о состоянии всех интерфесов, даже отключенных. Иначе команда конфигурирует указанный *интерфейс*.

Если первым после имени интерфейса идет имя поддерживаемого семейства адресов, это семейство адресов используется для декодирования и выдачи всех адресов протокола. В настоящее время поддерживаются семейства адресов `inet` (TCP/IP, используется по умолчанию), `inet6` (IPv6).

Синтаксис утилиты `ifconfig` различен для разных операционных систем, хотя эти различия и незначительны. Например, в операционной системе FreeBSD команда имеет следующий синтаксис:

```
ifconfig интерфейс inet IP-адрес netmask маска
```

Операционная система Solaris использует такой синтаксис:

```
ifconfig интерфейс inet IP-адрес/длина_префикса
```

Наконец, в операционной системе Linux допустимы следующие опции команды `ifconfig`:

```
ifconfig интерфейс inet IP-адрес netmask маска broadcast
широковещательный_адрес
```

Рассмотрим наиболее часто используемые опции команды, допустимые во всех реализациях операционной системы UNIX:

- *интерфейс* — имя интерфейса. В этом поле обычно указывают имя драйвера, за которым идет номер устройства, например, `eth0` для первого интерфейса Ethernet;
- `up` — позволяет активизировать интерфейс. Опция задается неявно при присвоении адреса интерфейсу;
- `down` — позволяет остановить работу драйвера для данного интерфейса;
- `mtu` *N* — этот параметр устанавливает максимальный размер пакета (Maximum Transfer Unit, MTU) для интерфейса;
- *адрес* — IP-адрес, присваиваемый интерфейсу.

Список интерфейсов, их текущие настройки и статус выводятся по команде `ifconfig`, введенной без параметров (в Solaris — с ключом `-a`).

Рассмотрим несколько примеров использования команды `ifconfig`. Если команда задается без опций, то выводится информация обо всех интерфейсах, присутствующих в системе. Вот как может выглядеть такая информация для операционной системы Linux:

```
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:10:4B:D1:D2:B4
          inet addr:194.15.28.10  Bcast:194.15.28.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:4
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:240 (240.0 b)
          Interrupt:11 Base address:0xc000
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:13055 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13055 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:890244 (869.3 Kb)  TX bytes:890244 (869.3 Kb)
```

Для получения информации по одному интерфейсу следует указать его имя в командной строке, как в этом примере:

```
# ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 00:0D:87:08:43:64
          inet addr:194.15.28.11  Bcast:194.15.28.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:240 (240.0 b)
          Interrupt:10 Base address:0x2000
```

В операционной системе Solaris для получения информации о сетевых интерфейсах нужно выполнить команду `ifconfig -a`:

```
# ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu
     8232 index 1 inet 127.0.0.1 netmask ff000000
rtls0: flags=1000803<UP,BROADCAST,MULTICAST,IPv4> mtu 1500 index 2
```

```
inet 194.15.28.10 netmask ffffff00 broadcast 194.15.28.255
ether 0:d:87:8:43:64
```

Каждый интерфейс системы должен иметь свой IP-адрес, который может быть присвоен такой командой:

```
# ifconfig eth1 194.15.28.40
```

В этом примере интерфейсу `eth1` присвоен адрес `194.15.28.40`. После установки параметров интерфейсов следует перезапустить демон `inetd` или перезагрузить систему. Проверку адреса можно выполнить с помощью команды `ping`, указав IP-адрес сетевого интерфейса:

```
# ping 194.15.28.40
PING 194.15.28.40 (194.15.28.40) 56(84) bytes of data.
64 bytes from 194.15.28.40: icmp_seq=1 ttl=64 time=0.039 ms
64 bytes from 194.15.28.40: icmp_seq=2 ttl=64 time=0.019 ms
```

Еще две опции команды `ifconfig` — `up` и `down` — позволяют запустить или остановить интерфейс, например:

```
# ifconfig eth1 up
# ifconfig eth0 down
```

В операционной системе Solaris для конфигурирования сетевого интерфейса необходимо указывать команду `ifconfig` с дополнительной опцией `plumb`, например:

```
# ifconfig eri0 plumb
```

Указанная опция позволяет связать сетевой интерфейс с устройством и инициализировать потоковые модули, требуемые для обмена данными по IP-протоколу. Для полной активизации сетевого интерфейса в Solaris следует ввести команду:

```
# ifconfig eri0 up plumb
```

Для того чтобы удалить сетевой интерфейс и закрыть все потоковые модули в Solaris, связанные с сетевым интерфейсом, используется опция `unplumb`:

```
# ifconfig eri0 up unplumb
```

Более подробно об особенностях команды `ifconfig` можно узнать из документации к операционным системам или из man-страниц.

8.3. Маршрутизация

Маршрутизация — это процесс передачи данных с компьютера, находящегося в одной сети, узлу-адресату, находящемуся в другой сети. Необходимость в маршрутизации возникает, например, при подключении компьютера к Ин-

тернету или к другой сети, поэтому практически каждый пользователь рано или поздно столкнется с проблемой настройки маршрутизации в операционной системе UNIX.

Функции маршрутизации выполняют специальные устройства — маршрутизаторы (routers), которые часто называют шлюзами. Маршрутизаторы соединяют две и более сетей и способны выполнять пересылку пакетов между ними (IP-forwarding). В общем случае все сети TCP/IP состоят из двух типов функциональных единиц — компьютеров (хостов) и маршрутизаторов. Во всех сетях обязательно присутствуют хосты, но далеко не в каждой есть маршрутизаторы. Функции маршрутизатора может выполнять как специальное устройство, так и обычный компьютер, имеющий две сетевые карты и настроенный соответствующим образом. Маршрутизаторы должны быть сконфигурированы таким образом, чтобы пересылать пакеты не только между соседними сетями, но и передавать пакеты узлам, не входящим в такие сети.

В простейшем случае две сети могут связываться между собой посредством одного маршрутизатора, как показано на рис. 8.4.

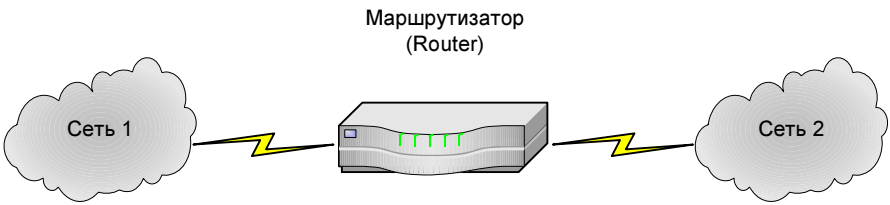


Рис. 8.4. Связь между двумя сетями посредством маршрутизатора

Как видно из рис. 8.4, сеть 1 связана с сетью 2 посредством маршрутизатора, что делает возможным обмен данными между этими сетями.

В более сложных случаях, когда необходимо связывать между собой несколько сетей, используется несколько маршрутизаторов, причем для доступа к одной и той же сети могут устанавливаться альтернативные маршруты, как показано на рис. 8.5.

Здесь маршрутизатор 3 напрямую соединяет сеть 1 и сеть 3. Если, предположим, сеть 2 становится недоступной, то обмен пакетами между сетью 1 и сетью 3 будет возможен благодаря маршрутизатору 3. Из этого рисунка видно, что некоторая избыточность конфигурации (а именно наличие маршрутизатора 3) позволяет улучшить надежность межсетевого взаимодействия. Естественно, что все сети в подобной конфигурации должны использовать одни и те же протоколы обмена, например, TCP/IP.

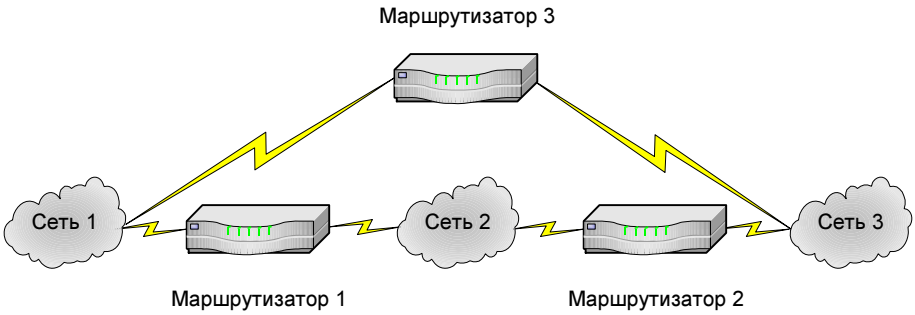


Рис. 8.5. Пример взаимодействия трех сетей посредством маршрутизаторов

Маршрутизаторы определяют дальнейший маршрут пакета, используя так называемые таблицы маршрутизации (routing tables). Эти таблицы содержат IP-адреса хостов и маршрутизаторов в тех сетях, с которыми данный маршрутизатор имеет связь, а также указатели на эти сети. При получении пакета маршрутизатор просматривает таблицу маршрутизации, пытаясь определить, содержит ли таблица IP-адрес, указанный в заголовке IP-пакета. Если маршрутизатор не обнаружит этот адрес, то он переправляет этот пакет другому маршрутизатору, указанному в таблице маршрутизации.

Процесс маршрутизации можно проиллюстрировать на рис. 8.6. Здесь показана топология трех сетей, соединенных двумя маршрутизаторами.

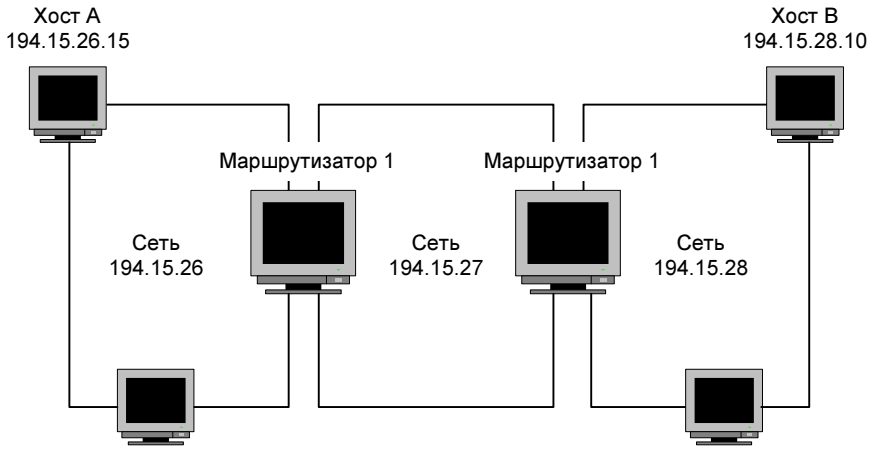


Рис. 8.6. Пример маршрутизации между тремя сетями

Маршрутизатор 1 соединяет сети 194.15.26 и 194.15.27, а маршрутизатор 2 соединяет сети 194.15.27 и 194.15.28. Предположим, что хост А, находящийся в сети 194.15.26, отправляет данные хосту В из сети 194.15.28. В этом случае выполняется следующая последовательность действий:

1. Хост А посылает пакет через сеть 194.15.26. Заголовок отправляемого пакета содержит IPv4-адрес получателя (194.15.28.10). Поскольку ни один из хостов сети 194.15.26 не имеет IPv4-адреса 194.15.28.10, то пакет принимает маршрутизатор 1.
2. Маршрутизатор 1 проверяет свои таблицы маршрутизации и обнаруживает, что ни один из компьютеров сети 194.15.27 не имеет адрес 194.15.28.10. Поскольку таблицы маршрутизации содержат адрес следующего маршрутизатора, каким является маршрутизатор 2, то именно ему маршрутизатор 1 и переправляет пакет.
3. Маршрутизатор 2 соединяет сети 194.15.27 и 194.15.28, поэтому располагает информацией о компьютере хост В. На последнем шаге маршрутизатор 2 передает пакет в сеть 194.15.28, и хост В его принимает.

Каким образом хост узнает, что нужно передавать пакеты в другую сеть, и как определяет, какой маршрутизатор для этого использовать? IP-адрес получателя, включаемый в заголовок пакета, определяет дальнейший маршрут пакета. Если IP-адрес узла доставки включает номер локальной сети, пакет будет доставлен адресату с данным IP-адресом, находящемуся в данной сети. Если же номер сети не соответствует локальной сети, то пакет направляется маршрутизатору, находящемуся в данной сети.

Для определения маршрута передачи используется протокол ARP (см. рис. 8.2), а сам процесс передачи иллюстрируется на рис. 8.7.

Предположим, что хост А, имеющий IP-адрес 194.15.26.2 и аппаратный (MAC) адрес 1 сетевого интерфейса, пытается передать пакет хосту В с IP-адресом 172.16.2.18 и MAC-адресом 22. Хочу сделать важное замечание: для доставки сетевого пакета узлу необходимо знать кроме IP-адреса также и его аппаратный адрес, или, как его чаще называют, MAC-адрес. MAC-адрес — это аппаратный адрес сетевого устройства (сетевой карты, маршрутизатора, хаба и т. д.), присваиваемый ему при изготовлении. Доставка пакета узлу невозможна, если неизвестен его аппаратный адрес.

Все последующие события можно представить следующей последовательностью шагов:

1. Хост А отправляет широковещательный ARP-запрос об аппаратном адресе узла с адресом 172.16.2.18. Если бы данный узел находился в сети 1, то он бы ответил на запрос хоста А, однако хост В находится в другой сети и ответа от него не будет.

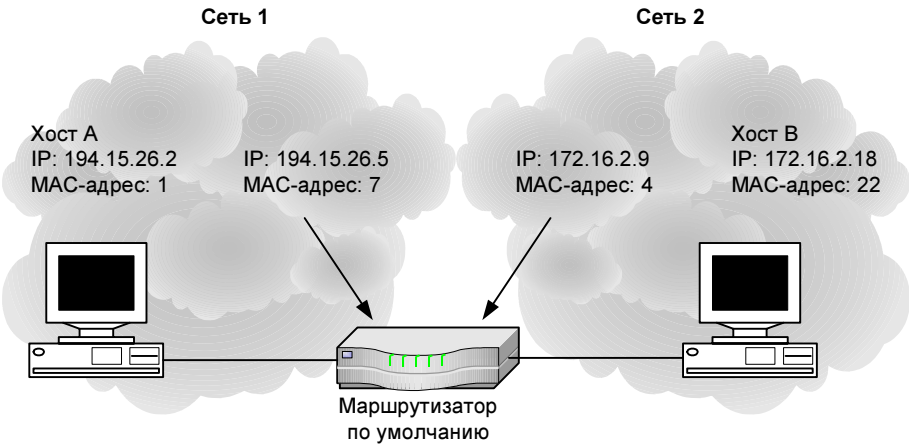


Рис. 8.7. Процесс передачи пакета посредством маршрутизатора

2. С помощью ARP хост A определяет MAC-адрес маршрутизатора по умолчанию. IP-адрес 194.15.26.5 маршрутизатора должен быть известен хосту A (он указывается при настройке сетевых параметров для данного хоста), но аппаратный адрес маршрутизатора должен определяться при помощи ARP.
3. Хост A отправляет пакет данных маршрутизатору с IP-адресом 194.15.26.5, причем заголовок каждого пакета содержит следующую информацию:
   - MAC-адрес отправителя: 1;
   - IP-адрес отправителя: 194.15.26.2;
   - аппаратный адрес узла-получателя: 7;
   - IP-адрес узла-получателя: 172.16.2.18.

Маршрутизатор, имеющий IP-адрес 194.15.26.5 и MAC-адрес 7, по заголовку пакета определяет, что пакет предназначен для дальнейшей передачи в сеть 172.16.2.

Маршрутизатор выполняет ARP-запрос для определения аппаратного адреса узла 172.16.2.18, причем аппаратный адрес сохраняется в кэше для последующего использования.

Маршрутизатор отправляет пакет в сеть 172.16.2, поместив следующую информацию в заголовок пакета:

- аппаратный адрес отправителя: 7;
- IP-адрес отправителя: 194.15.26.2;

- MAC-адрес получателя: 22;
- IP-адрес узла-получателя: 172.16.2.18;

Данные передаются по сети 172.16.2, причем сетевая карта узла адресата распознает свои IP- и MAC-адрес и получает данные.

Протокол ARP реализован в виде набора системных вызовов, используемых сетевыми программами для получения аппаратных адресов сетевых интерфейсов. Обширную информацию можно получить при помощи команды `arp`, которую можно запустить из командного интерпретатора shell.

Для настройки маршрутизации на уровне операционной системы можно воспользоваться одним из двух методов. Первый метод использует статические таблицы маршрутизации, маршруты в которые добавляются и удаляются вручную с помощью команды `route`. Статическая маршрутизация лучше всего подходит для небольших сетей, если существует только одна точка подключения к другим сетям и не существует запасных маршрутов (запасные маршруты могут быть использованы в случае выхода из строя основных). Если хотя бы одно из трех приведенных выше условий неверно, используется динамическая маршрутизация.

Второй метод имеет название динамической маршрутизации и предполагает выбор наилучшего маршрута из имеющихся. Его удобно использовать, если существуют два и более маршрута к определенному узлу сети. Здесь используются так называемые таблицы динамической маршрутизации, которые создаются и поддерживаются одним из протоколов динамической маршрутизации, программно реализованным в виде демона маршрутизации. Наиболее часто в UNIX-системах используются демоны маршрутизации `routed` и `gated`, которые обычно запускаются при старте системы и живут все время, пока система работает. Протоколы маршрутизации, используемые на конкретном хосте, определяют, как будет происходить обмен информацией о маршрутах с удаленными маршрутизаторами.

Данные о маршрутах включаются в таблицы динамической маршрутизации на основе информации, полученной от маршрутизаторов сети протоколом динамической маршрутизации. Кроме этого, протокол выбирает оптимальные маршруты доставки информации по указанному адресу. Естественно, что маршрутизаторы сети должны быть настроены для поддержки протокола динамической маршрутизации.

Маршрутизация IP выполняется на сетевом уровне (см. рис. 8.2) посредством обращения к таблице маршрутизации, которую демон маршрутизации обновляет примерно один раз каждые 30 секунд. Таблица маршрутизации обновляется при приеме ICMP-сообщения о необходимости перенаправления (ICMP

redirect). Обновление таблицы маршрутизации происходит при загрузке системы, позволяя установить некоторые исходные маршруты. Для просмотра таблиц маршрутизации предназначена команда `netstat`.

Сам просмотр таблицы маршрутизации можно представить в виде такой последовательности шагов:

1. Поиск совпадающего адреса хоста.
2. Поиск совпадающего адреса сети.
3. Поиск пункта по умолчанию. Пункт по умолчанию обычно указывается в таблице маршрутизации как сеть с идентификатором сети, равным нулю.

Совпавший адрес хоста используется всегда перед совпавшим адресом сети. Маршрутизация IP-пакетов — это поиск интерфейса, куда следует отправить пакет, в таблице маршрутизации, при этом демон маршрутизации обычно определяет политику маршрутизации.

Следует отличать маршрутизацию от протоколов маршрутизации: маршрутизация осуществляет передачу данных и базируется на информации, находящейся в таблицах маршрутизации, в то время как протоколы маршрутизации реализованы в виде программ, обновляющих информацию в таблицах маршрутизации. Можно настроить маршрутизацию так, что не будут использоваться никакие протоколы, но при этом придется создавать таблицы маршрутизации вручную.

Для получения информации о существующих маршрутах в операционной системе UNIX имеется несколько команд, из которых наиболее часто используются `route` и `netstat`. Следует сказать, что эти команды присутствуют во всех реализациях UNIX, хотя и имеют некоторые отличия, как в плане используемых опций, так и в форме выводимой информации.

Например, в операционной системе Solaris для получения информации о маршрутах применяется утилита `netstat`, а в Linux — `netstat` и `route`, причем `route` используется чаще.

Рассмотрим процесс настройки маршрутизации в UNIX и начнем со статической маршрутизации, когда маршруты к узлам назначения устанавливаются вручную командой `route`. Далее приведен пример добавления, изменения и удаления маршрута в операционной системе Solaris:

```
# route add 194.15.27.0 194.15.28.11 255.255.255.0
add net 194.15.27.0: gateway 194.15.28.11
# route change 194.15.27.0 194.15.28.12 255.255.255.0
change net 194.15.27.0: gateway 194.15.28.12
```

Здесь первая команда `route` устанавливает маршрут к сети 194.15.27.0 через маршрутизатор с адресом 194.15.28.11. Первым параметром этой команды является ключевое слово `add`, означающее, что нужно добавить маршрут. Следующим идет IP-адрес пункта назначения, маршрут к которому нужно установить. В данном случае таким пунктом назначения является сеть 194.15.27.0. Вместо IP-адреса сети можно указать ее имя (если оно указано в файле /etc/networks).

Замечу, что в качестве пункта назначения может выступать не только сеть, но и отдельный хост, который можно задать либо посредством его IP-адреса, либо указав его имя (если таковое присутствует в файле /etc/hosts). Следующим после пункта назначения указывается IP-адрес маршрутизатора (194.15.28.11).

В командной строке после адреса маршрутизатора можно (хотя и не обязательно) указать метрику маршрута (routing metric). Этот параметр может использоваться некоторыми операционными системами для оценки маршрута (является ли он прямым или используется внешний маршрутизатор).

Вторая команда `route` изменяет параметры маршрута, указывая другой IP-адрес (194.15.28.12) маршрутизатора. Первым параметром этой команды является ключевое слово `change`.

Проверить таблицу маршрутизации после выполнения этих двух команд несложно, выполнив команду `netstat`:

```
# netstat -r


Routing Table: IPv4
  Destination           Gateway           Flags  Ref   Use   Interface
-------------------------------------------------------------------------
194.15.28.0          yuryhost            U        1      0  rtls0
194.15.27.0          194.15.28.12        UG       1      0
224.0.0.0            yuryhost            U        1      0  rtls0
localhost            localhost           UH       4     88  lo0
```

Дополнительную информацию для пользователя представляют флаги, установленные в поле `Flags`. Здесь возможно пять различных вариантов:

- `U` — маршрут активен;
- `G` — маршрут подключен к шлюзу (маршрутизатору). Если флаг не установлен, то узел назначения подключен непосредственно;
- `H` — показывает, что маршрут ведет к хосту, а не к сети. Это означает, что в качестве пункта назначения используется IP-адрес хоста;

- `D` — маршрут был создан посредством перенаправления (ICMP Redirection);
- `M` — указывает на то, что маршрут был модифицирован, возможно, протоколом маршрутизации или посредством перенаправления.

Проанализируем полученный результат. В поле `Flags` отображается состояние маршрутов, и установленный флаг `U` свидетельствует о том, что все маршруты в данный момент активны. Флаг `G` указывает на то, что на данном маршруте используется маршрутизатор (194.15.28.12). Наконец, флаг `H` показывает, что маршрут ведет к хосту. В поле `Interface` показана привязка сетевых интерфейсов к IP-адресам.

Для удаления маршрута из таблицы маршрутизации нужно использовать команду `route` с ключевым словом `delete`:

```
# route delete 194.15.27.0 194.15.28.12
delete net 194.15.27.0: gateway 194.15.28.12
```

При настройке и проверке статической маршрутизации в операционной системе Linux можно использовать команду `route`. Например, для просмотра таблицы маршрутизации в Linux нужно выполнить команду `route` с опцией `-n`:

```
# route -n
Kernel IP routing table
Destination   Gateway      Genmask       Flags  Metric  Ref  Use Iface
194.15.27.0   194.15.28.12 255.255.255.0 U      0       0    0   eth0
127.0.0.0     0.0.0.0      255.0.0.0     U      0       0    0   lo
```

Как видно из результата, команда `route` предоставляет ту же информацию, что и `netstat` в Solaris. Кроме поля `Flags`, обе команды выводят и другую информацию. Например, в поле `Ref` выводится информация о количестве использований каждого маршрута, а поле `Use` показывает количество пакетов, прошедших по этому маршруту. Если, например, запустить программу `ping`, которая отправит 5 пакетов, значение счетчика станет равным 5 (при условии, что никто больше не использует этот маршрут). Последнее поле `Interface` отображает имя сетевого интерфейса.

Для добавления или удаления маршрутов также применяется команда `route`, но в операционной системе Linux необходимо использовать другие опции, чем в Solaris. Так, чтобы добавить маршрут к сети 194.15.27.0 через маршрутизатор 194.15.28.12, следует выполнить такую команду:

```
route add -net 194.15.27.0 netmask 255.255.255.0 gw 194.15.28.12 1
```

Во многих случаях требуется, чтобы маршруты устанавливались автоматически в момент загрузки системы. При использовании статической маршрутизации следует включить соответствующие командные строки в стартовый

командный файл. В операционных системах FreeBSD и Red Hat Linux таким командным файлом является rc.local (в Red Hat Linux полный путь /etc/rc.local), а в Solaris 9 — /etc/init.d/inetinit.

При работе в Solaris в конец файла /etc/init.d/inetinit нужно добавить строки примерно такого вида (естественно, что для вашей конфигурации IP-адреса будут иными):

```
route -n add default 194.15.28.1 > /dev/console
route -n add 194.15.26.0 194.15.28.3 > /dev/console
route -n add 194.15.27.0 194.15.28.3 > /dev/console
```

Здесь опция `-n` указывает `route`, что нужно отображать IP-адреса в виде чисел, а не имен. Без указания этой опции процесс загрузки может замедлиться, поскольку система попытается найти сервер DNS для разрешения имен. В операционной системе Linux эту опцию можно не использовать.

Поговорим более подробно о динамической маршрутизации. Динамическая маршрутизация используется для передачи информации между маршрутизаторами. Маршрутизаторы, используя протоколы маршрутизации, передают друг другу информацию о том, какие сети в настоящее время подключены к каждому из них. Пользовательский процесс, посредством которого маршрутизаторы обмениваются информацией, как мы знаем, называется демоном маршрутизации.

Динамическая маршрутизация реализуется посредством одного из протоколов динамической маршрутизации. К таким протоколам относится, в первую очередь, широко используемый RIP (Routing Information Protocol, протокол обмена информацией о маршрутизации), который присутствует практически во всех версиях TCP/IP. Кроме этого, достаточно широко применяются еще два протокола маршрутизации, OSPF (Open Shortest Path First) и BGP (Border Gateway Protocol).

Поскольку чаще других в операционных системах UNIX используется протокол RIP (Routing Information Protocol), то мы на нем и остановимся. Протокол RIP в большинстве операционных систем запускается демоном `routed` (в Solaris — `in.routed`). Демон маршрутизации обычно запускается во время загрузки системы и присутствует практически в каждой версии TCP/IP. Демон `routed` используется, в основном, в сетях малого и среднего размеров. Альтернативой `routed` является `gated`. Этот демон, кроме RIP, поддерживает и другие протоколы, например, IGP (Interior Gateway Protocol) и EGP (Exterior Gateway Protocol). В большинстве систем, которые используют демоны маршрутизации, запускается `routed`, однако при необходимости поддерживать разные протоколы маршрутизации используется `gated`.

Демон маршрутизации отвечает за политику маршрутизации (routing policy), выбирая, какие маршруты необходимо включить в таблицу маршрутизации. Если демон обнаружил несколько маршрутов к пункту назначения, он выбирает каким-либо образом лучший маршрут, и именно этот маршрут добавляет в таблицу маршрутизации. Если демон определил, что канал не существует или исчез (возможно, из-за неисправности маршрутизатора), он может удалить соответствующие маршруты или добавить альтернативные, чтобы обойти возникшую неисправность.

Динамическая маршрутизация не меняет способы, с помощью которых ядро осуществляет маршрутизацию на IP-уровне. Ядро просматривает свою таблицу маршрутизации, отыскивая маршруты к хостам, маршруты к сетям и маршруты по умолчанию. Меняется только способ помещения информации в таблицу маршрутизации — вместо запуска команды `route` или использования загрузочных файлов маршруты добавляются и удаляются динамически демоном маршрутизации, который работает постоянно.

Необходимо учитывать, что маршрутизаторы сети с динамической маршрутизацией должны поддерживать протокол RIP, иначе процесс обмена данными между ними и RIP будет невозможен, что сделает применение протокола бесполезным.

В настоящее время ощущается нехватка адресов класса B, поэтому узлам с несколькими сетями приходится присваивать несколько идентификаторов сетей класса C вместо одного идентификатора сети класса B. С одной стороны, появление адресов класса C решает проблему переполнения количества адресов класса B. С другой стороны, появляется еще одна проблема: каждая сеть класса C требует записи в таблице маршрутизации. Бесклассовая маршрутизация между доменами (Classless Interdomain Routing, CIDR) позволяет свести к минимуму рост таблиц маршрутизации в Интернете.

Основная концепция, заложенная в CIDR, заключается в том, что несколько IP-адресов можно расположить определенным образом, позволяющим уменьшить количество записей в таблице маршрутизации. Термин "бесклассовый" используется потому, что решения о маршрутизации принимаются на основе масок, накладываемых на полный 32-битовый IP-адрес.

Очень важной задачей, помимо установки требуемых маршрутов, является проверка их доступности, а также поиск и устранение неисправностей. Хорошо, если поиск в таблице маршрутизации заканчивается успешно, и будет обнаружено соответствие хотя бы с одним маршрутом. Что произойдет, если маршрут по умолчанию отсутствует, и не будет найдено совпадение с указанным пунктом назначения? Для выявления подобных неисправностей в таких случаях можно использовать команды `ping`, `traceroute` и `netstat`. Мы рассмотрим методику диагностирования сетей далее в этой главе.

8.4. Статистика работы сети

Контроль функционирования сети и сетевых интерфейсов в операционной системе UNIX осуществляется с помощью утилиты `netstat`. В разных модификациях она присутствует во всех реализациях UNIX, позволяя отобразить содержимое различных структур данных, связанных с сетью, в разных форматах в зависимости от указанных опций.

Если утилита используется в форме

```
netstat [-Aan] [-f семейство_адресов]
        [-I интерфейс] [-p имя_протокола] [система] [core]
```

то она отображает список активных сокетов для каждого протокола.

Вторая форма программы `netstat`

```
netstat [-n] [-s] [-i | -r]
        [-f семейство_адресов]
        [-I интерфейс] [-p имя_протокола] [система] [core]
```

выбирает одну из нескольких других сетевых структур данных и отображает ее на консоли.

Третья форма

```
netstat [-n] [-I интерфейс] интервал [система] [core]
```

показывает динамическую статистику пересылки пакетов по сконфигурированным сетевым интерфейсам. Аргумент *интервал* определяет количество секунд, в течение которых выполняется сбор информации между двумя последовательными выводами результата. Для аргумента *система* значением по умолчанию является /unix; для аргумента `core` в качестве значения по умолчанию используется /dev/kmem.

Вот смысл опций команды `netstat`:

- `-a` — отображает состояние всех сокетов, кроме тех, что используются серверами;
- `-A` — выводит адреса любых управляющих блоков протокола, связанных с сокетами, и предназначена, в основном, для отладки;
- `-i` — отображает состояние автоматически сконфигурированных интерфейсов. При этом состояние интерфейсов, статически сконфигурированных в системе, но не обнаруженных во время загрузки, не выводится;
- `-n` — отображает сетевые адреса как числа вместо их символьного представления, используемого по умолчанию. Опция может быть использована с любым форматом вывода;

- `-r` — выводит таблицы маршрутизации. Заданная вместе с опцией `-s`, показывает статистику маршрутизации;
- `-s` — отображает статистическую информацию по протоколам. Заданная вместе с опцией `-r`, показывает статистику маршрутизации;
- `-f` *семейство_адресов* — ограничивает вывод статистики или адресов управляющих блоков только указанным семейством адресов:
  - `inet` — для семейства адресов AF_INET;
  - `unix` — для семейства адресов AF_UNIX;
- `-I` *интерфейс* — позволяет выводить информацию об указанном сетевом интерфейсе в отдельной колонке;
- `-p` *имя_протокола* — позволяет ограничить вывод статистики или адресов управляющих блоков только протоколом с указанным именем, например, `tcp`.

Команда `netstat` для каждого активного сокета выдает сведения о протоколе, размере очередей приема, локальном и удаленном адресах, а также о состоянии протокола.

Символьный формат обычно применяется для отображения адресов сокетов и выводится в форме:

хост.порт

если имя хоста указано, либо:

сеть.порт

если адрес сокета задан сетью, а не конкретным хостом. Имена хостов и сетей берутся из соответствующих записей в файле /etc/hosts или /etc/networks.

Если имя сети или хоста неизвестно (или если задана опция `-n`), адрес отображается в числовом виде, при этом неуказанные адреса и порты выводятся в виде символа `*`. Состояния сокетов кодируются следующими обозначениями:

- `CLOSED` — сокет закрыт;
- `LISTEN` — ожидание входящих соединений;
- `SYN_SENT` — попытка установить соединение;
- `SYN_RECEIVED` — выполняется начальная синхронизация соединения;
- `ESTABLISHED` — соединение установлено;
- `CLOSE_WAIT` — удаленная сторона отключилась, ожидание закрытия сокета;
- `TIME_WAIT` — ожидание после закрытия и подтверждения отключения удаленной стороны.

Вывод результатов во второй форме команды `netstat` зависит от выбора опции: `-i` или `-r`. Если указаны обе опции, `netstat` использует `-i`. В этом случае отображаются все имеющиеся маршруты и статус каждого из них. Каждый маршрут состоит из целевого хоста или сети и шлюза (gateway), который используется для пересылки пакетов. Столбец `FLG` (флаги) показывает статус маршрута.

Вот примеры выполнения команды `netstat` с различными опциями в операционной системе Linux:

```
# netstat -i
Kernel Interface table
Iface MTU Met RX-OK RX-ERR RX-DRP RX-OVR  TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0  1500 0   0     0      0      0       9      0      0      0    BMU
eth1  1500 0   0     0      0      0       17     0      0      0    BMRU
lo   16408 0   91    0      0      0      988     0      0      0    LRU
```

Информация о прослушивающих сокетах:

```
# netstat -l
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address          Foreign Address      State
tcp        0      0 *:32768                *:*                  LISTEN
tcp        0      0 yuryhost:32769         *:*                  LISTEN
tcp        0      0 *:imaps                *:*                  LISTEN
tcp        0      0 *:pop3s                *:*                  LISTEN
tcp        0      0 *:time                 *:*                  LISTEN
```

Для получения статистической информации по протоколам можно использовать команду `netstat` с опцией `-s`:

```
# netstat -s
Ip:
    7283 total packets received
    1 forwarded
    0 incoming packets discarded
    4149 incoming packets delivered
    4192 requests sent out
Icmp:
    3 ICMP messages received
    0 input ICMP message failed.
    ICMP input histogram:
        destination unreachable: 3
```

```
    3 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        destination unreachable: 3
Tcp:
    40 active connections openings
    110 passive connection openings
    0 failed connection attempts
    0 connection resets received
    2 connections established
    729 segments received
    722 segments send out
    0 segments retransmited
    0 bad segments received.
    0 resets sent
```

Следующий пример показывает формат вывода информации о сетевых соединениях в операционной системе Solaris:

```
# netstat -a

UDP: IPv4
Local Address      Remote Address     State
-------------------- -------------------- -------
 *.route                              Idle
 *.sunrpc                             Idle
 *.*                                  Unbound
 *.39265                              Idle
 *.*                                  Unbound
UDP: IPv6
Local Address      Remote Address      State      If
---------------------- ------------------------------ ---------- -----
 *.177                                Idle

TCP: IPv4
Local Address      Remote Address    Swind Send-Q Rwind Recv-Q  State
------------------------------------ ----- ------ ----- ------ -------
*.telnet             *.*                0      0 49152      0 LISTEN
*.ftp                *.*                0      0 49152      0 LISTEN
*.finger             *.*                0      0 49152      0 LISTEN
```

```
*.login              *.*                0      0 49152      0 LISTEN
*.shell              *.*                0      0 49152      0 LISTEN
. . .
SCTP:
Local Address    Remote Address  Swind  Send-Q Rwind  Recv-Q StrsI/O
State
------------------------------ ----- ------ ------ ------- -----------
0.0.0.0          0.0.0.0      0      0 102400      0  32/32  CLOSED

Active UNIX domain sockets
Address    Type          Vnode     Conn       Local Addr      Remote Addr
d612a8a0   stream-ord   00000000     00000000
d612ac18   stream-ord   00000000     d4ee8b40 /tmp/.X11-unix/X0
```

Как видно из этих примеров, `netstat` является очень информативной командой и позволяет получить довольно подробную статистику по используемым протоколам и соединениям.

8.5. Диагностика сети и поиск неисправностей

Проверка работоспособности сетевых настроек в операционной системе UNIX — задача, с которой сталкивается любой пользователь, чей компьютер подключен к сети. Современные операционные системы имеют, как правило, развитую диагностику, позволяющую решить многие проблемы, связанные с работоспособностью сети. Тем не менее, включенные в состав UNIX классические утилиты для проверки сетей позволяют выполнить большинство проверок, причем весьма эффективно. К таким утилитам следует отнести программы `ping` и `traceroute`.

Программа `ping` выполняет проверку доступности удаленного хоста, посылая специальные пакеты `ECHO_REQUEST` протокола ICMP и ожидая от них ответа. Команда имеет два варианта реализации:

```
/usr/sbin/ping хост [тайм-аут]
/usr/sbin/ping -s [-drvRlfnq] [-i время_ожидания] [-p шаблон]
                  хост [размер_данных [кол_пакетов]]
```

Послав дейтаграмму `ECHO_REQUEST`, программа `ping` ожидает ответа `ICMP ECHO_RESPONSE` от указанного хоста или сетевого шлюза. Если хост отвечает, `ping` выдает сообщение о его доступности ("host is alive") в стандартный выходной поток и завершает работу. В противном случае по истечении опреде-

ленного интервала времени (тайм-аута) `ping` выдает сообщение о недоступности хоста ("no answer from host"). Стандартное значение тайм-аута равно 20 секундам.

Если указана опция `-s`, утилита `ping` посылает дейтаграмму каждую секунду и отображает одну строку вывода для каждого полученного ответа `ECHO_RESPONSE`. В этом случае `ping` вычисляет времена обхода (round trip times) и статистику потери пакетов. После завершения или по истечении тайм-аута команда отображает итоговую информацию.

Если указана необязательная опция *`кол_пакетов`*, то `ping` посылает указанное количество запросов хосту или шлюзу, после чего прекращает работу. Если количество пакетов не указано, команда будет выполняться бесконечно и прервать ее можно нажатием клавиши <Delete>.

Стандартный размер посылаемого пакета дейтаграммы равен 64 байтам, но можно изменить его, указав параметр *`размер_данных`*. Следует учитывать, что `ping` автоматически добавляет 8-байтовый заголовок к каждой посылаемой дейтаграмме, поэтому размер пакета, отображаемый при использовании опции `-s` с аргументом *`размер_данных`*, всегда будет на 8 байтов больше, чем указанное значение.

Перед тем как проверить работоспособность удаленного хоста, следует убедиться в работоспособности сетевого интерфейса на локальной машине, для чего выполнить команду `ping`, указав IP-адрес сетевого интерфейса на локальном хосте.

Рассмотрим более подробно опции команды `ping`:

- `-d` — режим отладки. Поставщику передается опция `SO_DEBUG`;
- `-f` — лавинный `ping`. Выдает пакеты сразу после возвращения, причем для каждого посланного `ECHO_REQUEST` печатается точка ".", а для каждого полученного `ECHO_REPLY` печатается "забой" (Backspace). Это позволяет быстро оценить процент потери пакетов, но при этом существенно увеличивается загрузка сети, поэтому использовать ее надо осторожно;
- `-i` *`время_ожидания`* — устанавливает время ожидания в секундах между посылками пакетов. По умолчанию этот интервал равен одной секунде. Эту опцию нельзя использовать с опцией `-f`;
- `-p` *`шаблон`* — указанный шаблон используется для заполнения посылаемых пакетов определенными данными. Шаблон задается, как шестнадцатеричная строка байтов длиной до 16 байтов, и повторяется для заполнения раздела данных пакета. Например, указание опции `-p ff` вызывает заполнение пакетов единицами. Опция полезна при поиске проблем сети, связанных с передаваемыми данными;

- `-q` — сокращенный вывод. Не выдается ничего, кроме суммарных строк при запуске и завершении работы;
- `-r` — не использовать обычные таблицы маршрутизации, посылая пакеты напрямую указанному хосту в подключенной сети. Если хост не находится в непосредственно подключенной сети, возвращается ошибка. Опция используется для проверки локального хоста через интерфейс, удаленный демоном маршрутизации;
- `-R` — записывать маршрут в заголовок IP-пакета. В этом случае содержимое записи маршрута выдается на консоль, если указана опция `-v`;
- `-s` — посылать дейтаграмму каждую секунду и отображать строку вывода для каждого полученного ответа `ECHO_RESPONSE` (при отсутствии ответа ничего не выдается);
- `-v` — детальный вывод. Выдает все полученные пакеты ICMP, кроме `ECHO_RESPONSE`.

В следующем примере проверяется доступность хоста **www.perl.org**:

```
# ping www.perl.org
PING x2.develooper.com (198.182.196.56) 56(84) bytes of data.
64 bytes from x2.develooper.com (63.251.223.172): icmp_seq=1 ttl=50
time=419 ms
. . .
64 bytes from x2.develooper.com (63.251.223.172): icmp_seq=9 ttl=50
time=409 ms
64 bytes from x2.develooper.com (63.251.223.172): icmp_seq=10 ttl=50
time=379 ms


--- x2.develooper.com ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9329ms
rtt min/avg/max/mdev = 359.982/391.913/419.979/19.345 ms
```

Хочу заметить, что стандартный вариант команды `ping` имеет весьма ограниченные возможности по проверке доступности или существования удаленного хоста. Команда использует ICMP-протокол, а этого часто бывает недостаточно для того, чтобы определить доступность хоста для работы, например, с протоколом UDP.

Удаленный компьютер может быть настроен на работу только с TCP, UDP или иными протоколами, поэтому в таких случаях следует применять иные средства для проверки работоспособности сетевых компьютеров.

Для тестирования сетей очень полезной является утилита `traceroute`. При более-менее серьезных проблемах без этой программы не обходится никто.

Программа `traceroute` позволяет выявлять последовательность шлюзов, через которые проходит IP-пакет на пути к пункту своего назначения. Команда допускает много опций, большинство из которых используются редко. Синтаксис команды можно представить в таком виде:

```
traceroute имя_машины
```

Опция *имя_машины* может быть задана в символической или числовой форме, а выходная информация представляет список машин, начиная с первого шлюза и заканчивая пунктом назначения. Кроме того, команда `traceroute` показывает полное время прохождения каждого шлюза, позволяя просмотреть маршрут, по которому двигаются IP-дейтаграммы от одного хоста к другому.

Команда `traceroute` устанавливает поле времени жизни (число переходов) исходящего пакета так, чтобы это время истекало до достижения пакетом пункта назначения. По истечении времени жизни текущий шлюз отправит сообщение об ошибке на машину-источник. Каждое приращение поля времени жизни позволяет пакету пройти на один шлюз дальше.

Команда `traceroute` посылает для каждого значения поля времени жизни три пакета. Если промежуточный шлюз распределяет трафик по нескольким маршрутам, то пакеты могут возвращаться разными хостами, и их значения все будут выведены на консоль. Некоторые системы не посылают уведомлений о пакетах, время жизни которых истекло, другие посылают уведомления, поступающие на машину-источник после того, как истекло время их ожидания командой `traceroute`. Такие шлюзы обозначаются рядом символов `*`. Если конкретный шлюз определить нельзя, команда `traceroute` в большинстве случаев все же определит следующие за ним узлы маршрута.

Рассмотрим пример работы команды `traceroute`:

```
# traceroute kernig
traceroute to kernig (192.252.13.65), 30 hops max, 40 byte packets
1 richi (140.252.13.35)  20 ms  10 ms  10 ms
2 kernig (140.252.13.65)  110 ms  110 ms  110 ms
```

Первая строка без номера содержит имя и IP-адрес пункта назначения и указывает на то, что величина TTL (Time-To-Live, время жизни пересылаемого пакета) не может быть больше 30, размер дейтаграммы установлен в 40 байтов, из которых 20 байтов отводится на IP-заголовок, 8 байтов на UDP-заголовок и 12 байтов на пользовательские данные. Следует отметить, что в 12 байтах пользовательских данных сохраняется номер последовательности, который увеличивается на единицу при отправке следующей дейтаграммы, копия исходящего TTL и время, когда дейтаграмма была отправлена.

В следующих двух строках вывод начинается с TTL, затем указывается имя хоста или маршрутизатора и их IP-адреса. Для каждого значения TTL отправляются 3 дейтаграммы, для каждого возвращенного ICMP-сообщения рассчитывается и отображается время возврата (round-trip). При отсутствии ответа в течение пяти секунд на любую из трех дейтаграмм выводится звездочка, затем отправляется следующая дейтаграмма. В данном примере первые три дейтаграммы имели TTL, установленные в единицу, а ICMP-сообщения вернулись через 20, 10 и 10 миллисекунд. Следующие три дейтаграммы были отправлены с TTL, равным 2, а ICMP-сообщения вернулись с задержкой 110 миллисекунд. Поскольку TTL со значением 2 достигло конечного пункта назначения, программа прекратила свою работу.

Команда `traceroute` имеет некоторые особенности, которые следует учитывать при диагностике сетей:

- не существует гарантии, что маршрут, который используется в данный момент, будет использоваться и дальше, даже если две последовательные дейтаграммы были отправлены к одному и тому же пункту назначения;
- не существует гарантии того, что путь, по которому вернется ICMP-сообщение, совпадет с путем, по которому `traceroute` отправила дейтаграмму. Это означает, что время возврата, отображенное программой, может не совпадать со временем, необходимым для передачи исходящей дейтаграммы и возвращения сообщения. Возможен вариант, что UDP-дейтаграмма дойдет от источника до маршрутизатора за 1 секунду, однако ICMP-сообщение проделает обратный путь за 3 секунды, при этом время возврата будет напечатано как 4 секунды.

Кроме рассмотренных программ в различных версиях операционных систем UNIX имеются собственные утилиты для настройки и диагностики сетей, такие, например, как команды `sysctl` (FreeBSD, Linux) и `ndd` (Solaris).

В приведенных далее примерах первая команда выводит значение параметра, отвечающего за ретрансляцию дейтаграмм, а вторая команда присваивает этому параметру значение 1, что означает разрешение ретрансляции. Для Linux командная строка выглядит так:

```
sysctl net.ipv4.ip_forward
sysctl -w net.ipv4.ip_forward=1
```

а для FreeBSD принимает вид:

```
sysctl net.inet.ip.forwarding
sysctl net.inet.ip.forwarding=1
```

Те же самые операции в операционной системе Solaris можно выполнить с помощью команды `ndd`:

```
ndd /dev/ip ip_forwarding
ndd -set /dev/ip ip_forwarding 1
```

8.6. Сетевые сервисы UNIX

Операционная система UNIX включает целый ряд сетевых сервисов или, по-другому, служб, которые позволяют системе взаимодействовать с другими компьютерами в сети на уровне приложений пользователя (telnet, FTP, SMTP), автоматизировать сетевые настройки компьютеров в сети (DHCP, DNS), а также получать доступ к ресурсам других хостов (NFS). Общие принципы функционирования сетевых сервисов были рассмотрены ранее в этой главе, сейчас же мы подробно проанализируем работу некоторых из них, а именно: DNS, NFS и DHCP. Эти сервисы являются чрезвычайно важными для функционирования компьютера в сети, и без понимания их работы правильно выполнить настройку сетевых параметров даже на отдельном хосте невозможно.

Анализ функционирования сетевых сервисов начнем с DNS.

8.6.1. Служба имен DNS

Система имен доменов DNS (Domain Name System) представляет собой распределенную базу данных, которая используется приложениями TCP/IP для установления соответствия между именами хостов и IP-адресами. Кроме того, DNS используется для маршрутизации электронной почты и для работы в Интернете.

Доступ к распределенной базе данных осуществляется посредством так называемых доменных имен (domain names). Каждое из доменных имен представляет собой не что иное, как путь в большом дереве, известном под названием пространство доменных имен (domain name space). Дерево представляет собой иерархическую структуру, подобную той, которую имеет файловая система UNIX (рис. 8.8).

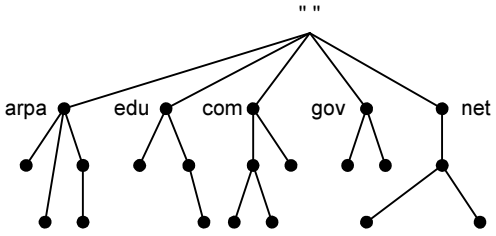


Рис. 8.8. Пространство доменных имен

В вершине дерева расположен корневой узел дерева — специальный узел без метки. В пространстве доменных имен допускается вложенность до 127 уровней, хотя на практике достичь такого уровня вложенности невозможно.

Каждый узел дерева имеет текстовую метку, длина которой не превышает 63 символов, причем метка нулевой длины зарезервирована для корня. Полное доменное имя любого узла представляет собой последовательность меток, расположенных на пути от данного узла к корню. Доменные имена всегда читаются от данного узла по направлению к корню (вверх по дереву), причем составляющие этих имен разделяются точками, как это показано на рис. 8.9.

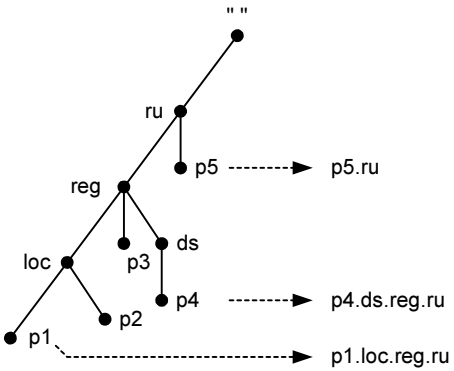


Рис. 8.9. Пример формирования доменных имен

Метки могут содержать как прописные, так и строчные литеры. Имя домена, которое заканчивается точкой, называется абсолютным именем домена (absolute domain name) или полным именем домена (Fully Qualified Domain Name, FQDN). Примером такого имени является **sun.tuc.noao.edu**. Если имя домена не заканчивается точкой, подразумевается, что имя должно быть завершено, причем зависит это от используемого программного обеспечения DNS.

Наряду с полными доменными именами хостов очень важным является термин "домен". Доменом является поддерево пространства доменных имен, причем именем домена является имя верхнего узла в поддереве (рис. 8.10).

Имя домена (domain name) для любого узла в дереве — это последовательность меток, которая начинается с узла, выступающего в роли корня, при этом метки разделяются точками. Каждый узел дерева должен иметь уникальное имя домена, однако одинаковые метки могут быть использованы в различных точках дерева.

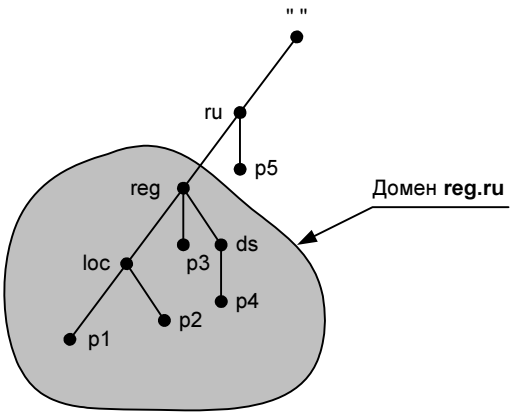


Рис. 8.10. Пример домена

Домены верхнего уровня поделены на три зоны и имеют определенное назначение, например:

- **arpa** — специальный домен, используемый для сопоставления "`адрес-имя`";
- семь 3-символьных доменов, которые называются общими или, в некоторых случаях, организационными;
- 2-символьные домены, основанные на кодах стран и описанные в ISO 3166. Они называются доменами стран или географическими доменами.

Многие 2-символьные домены стран второго уровня очень похожи на основные домены: домен **ac.uk**, например, принадлежит академическим институтам, а **co.uk** — коммерческим организациям Великобритании.

Своим названием "распределенная" DNS обязана способу реализации — на одном узле Интернета хранится только часть информации, причем каждый узел поддерживает собственную информационную базу данных, управляемую программой-сервером, которая может отправить запрос по Интернету к другим системам. Обобщая, можно сказать, что DNS предоставляет протокол, который позволяет клиентам и серверам общаться друг с другом.

В ряде случаев следует определиться, нужно ли устанавливать службу DNS на вашем хосте — особенно это касается довольно сложной настройки сервера имен. Кроме DNS, в операционной системе, которую вы используете, можно применить и другой механизм для разрешения имен, например, статические таблицы имен. Для всех операционных систем статические таблицы имен содержатся в файле /etc/hosts, иногда (при использовании сетевой фай-

ловой системы NFS) — в файле /etc/exports. При поиске соответствия имен IP-адресам часто используется файл /etc/nsswitch.conf, позволяющий определить, где искать информацию.

Статические таблицы имен часто используются на хостах, не входящих в сети, например, на домашних компьютерах. При этом пользователь может редактировать содержимое файла /etc/hosts, который хранит записи вида

IP-адрес имя_хоста имя_хоста . . .

Вот пример содержимого файла /etc/hosts в Linux:

```
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1      yuryhost   localhost.localdomain   localhost
194.15.28.10   host1
```

В этом файле перечислены соответствия IP-адресов и имен, которые известны компьютеру непосредственно, без обращения к DNS. В небольших изолированных сетях преобразования "имя — IP-адрес" и "IP-адрес — имя" можно выполнять без помощи DNS, используя только файл /etc/hosts.

Помимо файла /etc/hosts, для разрешения имен в UNIX может использоваться файл /etc/nsswitch.conf, который требуется для нормальной работы других служб, например, NIS (Network Information Service). Смысл применения этого файла состоит в том, что для большинства UNIX-систем требуется указать, какие службы и в каком порядке будут вызываться для распознавания имен, и такие инструкции определены в /etc/nsswitch.conf в строке `hosts`. Формат строки:

`hosts:` *служба* [*модификатор(ы)*] *служба* ...

Здесь *служба* — служба распознавания имен (`dns`, `files`, `nis`), причем `files` означает просмотр /etc/hosts. Службы используются в порядке их появления в строке, модификатор задает условие перехода от использования одной службы к другой и имеет вид:

событие=действие

Вот примеры событий:

- `NOTFOUND` — запрос обслужен, данные не найдены;
- `TRYAGAIN` — сервис временно недоступен (например, произошел таймаут);
- `UNAVAIL` — сервис недоступен (например, не сконфигурирован, отсутствует файл resolv.conf);

- `SUCCESS` — запрос обслужен, данные найдены;
- `Continue` — использовать следующую службу;
- `Return` — прекратить поиск.

Модификаторов может быть несколько, тогда они следуют друг за другом внутри квадратных скобок через пробел. Пример строки наиболее распространенной конфигурации:

```
hosts: files [NOTFOUND=continue] dns
```

Это означает, что сначала просматривается файл /etc/hosts, и в случае, если искомые данные не найдены, запрашивается сервер DNS.

В Linux порядок использования служб распознавания имен задается в файле /etc/host.conf, например:

```
order hosts,bind,nis
multi on
```

Следует сказать, что для разрешения имен во многих случаях используется комбинация статических таблиц и службы DNS. Например, при начальной загрузке операционной системы требуется знать IP-адреса серверов до того, как стартует служба DNS, поэтому здесь удобно использовать информацию из файла /etc/hosts.

Установка сервисов DNS может понадобиться в таких случаях:

- при необходимости постоянной работы с ресурсами Интернета, особенно учитывая то обстоятельство, что почти все интернет-сервисы (WWW, электронная почта, FTP) используют DNS. При этом вовсе не обязательно выполнять полную настройку DNS, тем более что провайдер услуг Интернета такую настройку выполняет самостоятельно;
- при использовании собственной TCP/IP-сети, особенно, если сеть включает в себя несколько десятков или сотен хостов.

Напомню, что для пользователя любой сетевой сервис представляет собой, как минимум, две взаимосвязанные части — серверную, выполняющую прием запросов на обслуживание, и клиентскую, которая посылает по сети запросы на обслуживание. В данном случае сервер службы имен (сервер DNS) принимает запросы от клиентских хостов на разрешение имен удаленных компьютеров, выполняет поиск соответствия имен хостов их IP-адресам в распределенной базе данных DNS и пересылает клиенту ответ.

Клиентское приложение получает доступ к DNS посредством разборщика (resolver) — процедуры, выполняющей создание, отправку и интерпретацию пакетов, используемых серверами имен Интернета. В UNIX-системах прило-

жение может получить доступ к разборщику посредством двух библиотечных функций `gethostbyname()` и `gethostbyaddr()`, которые компонуются с приложением во время его создания. Первая из этих функций принимает в качестве аргумента имя хоста и возвращает его IP-адрес, а вторая воспринимает в качестве аргумента IP-адрес и возвращает имя хоста. Для разрешения имен разборщик подключается к одному или нескольким серверам DNS. Процедура-разборщик не включена в ядро операционной системы, как это сделано для протоколов TCP/IP. Приложению необходимо преобразовать имя хоста в IP-адрес перед открытием соединения или послать дейтаграмму с использованием UDP.

Одна важная характеристика DNS — это передача ответственности внутри DNS. Не существует организации, которая бы управляла и обслуживала все дерево в целом и каждую метку в отдельности. Вместо этого одна организация (NIC) обслуживает только часть дерева (домены верхнего уровня), а ответственность за определенные зоны передает другим организациям.

Зона (zone) — это отдельно администрируемая часть дерева DNS, например, домен второго уровня **noao.edu** представляет отдельную зону. Многие домены второго уровня могут быть поделены на меньшие зоны, например, университет может поделить свою зону на подзоны по факультетам, а домен компании может поделиться на зоны по принципу организационных единиц (филиалов или отделов).

Сервер DNS может обслуживать одну или несколько зон, хотя, например, для одной зоны можно использовать основной сервер DNS (primary name server) и один или несколько вторичных серверов DNS (secondary name servers). Первичный и вторичный серверы должны быть независимы и избыточны таким образом, чтобы система DNS не вышла из строя при отказе одного из серверов.

Основное отличие между первичными и вторичными серверами заключается в том, что первичные загружают всю необходимую информацию из дисковых файлов, тогда как вторичные получают информацию от первичного. Процесс передачи информации от первичного сервера вторичному называется передачей зоны (zone transfer). Когда в зоне появляется новый хост, администратор добавляет соответствующую информацию (как минимум, имя и IP-адрес) в дисковый файл на первичном сервере. После чего первичный сервер DNS уведомляется о необходимости повторно считать свои конфигурационные файлы. Вторичные серверы регулярно опрашивают первичные (обычно каждые 3 часа), и если первичные содержат новую информацию, вторичные получают ее с использованием передачи зоны.

Что произойдет, если сервер DNS не содержит необходимой информации? В этом случае он должен установить контакт с другим сервером DNS, но

здесь может возникнуть еще одна проблема — далеко не каждый сервер DNS знает, как обратиться к другому серверу. Проблема решается таким образом: каждый сервер DNS обладает информацией о том, как установить контакт с корневыми серверами DNS (root name servers).

Первичные серверы должны знать IP-адреса корневых серверов, а не их DNS-имена. Корневой сервер, в свою очередь, знает имена и IP-адреса каждого сервера DNS для всех доменов второго уровня. При этом возникает последовательный процесс: запрашивающий сервер устанавливает контакт с корневым сервером, который сообщает запрашивающему серверу о необходимости обратиться к другому серверу и т. д.

Фундаментальная характеристика DNS — это кэширование (caching). Когда DNS-сервер получает информацию о соответствии, скажем, IP-адресов именам хостов, он кэширует эту информацию таким образом, что в случае следующего запроса может быть использована информация из кэша, дополнительный запрос на другие серверы не делается.

Серверы имен могут быть либо рекурсивными, либо нерекурсивными (итеративными), в зависимости от того, как они обрабатывают запросы. Рекурсивный сервер должен вернуть либо ответ на запрос, либо сообщение об ошибке, при этом все действия по поиску данных и опросу других серверов сервер берет на себя.

Что же касается нерекурсивного сервера, то здесь ситуация несколько иная. Если нерекурсивный сервер хранит в кэше информацию от предыдущего запроса (IP-адрес), то он может ее использовать при поступлении очередного запроса. Если же такая информация отсутствует, то нерекурсивный сервер в качестве ответа возвращает клиенту адрес главного (корневого) сервера имен в другом домене, предполагая, что сделавший запрос клиент перенаправит этот запрос указанному серверу. Клиент нерекурсивного сервера должен уметь обрабатывать такие пересылки. Нерекурсивными серверами обычно являются корневые серверы, поскольку из-за большой загрузки они не в состоянии выполнить полностью обработку запроса.

Процесс обработки запроса проиллюстрирован на рис. 8.11.

Предположим, что хост **c1.priv.net** хочет узнать IP-адрес узла **dc2.loc.comm.org**. Для этого он запрашивает информацию у своего локального сервера имен **ns.priv.net** (1). Если локальный сервер имен не знает нужный адрес, то он может обратиться к корневому серверу имен, обслуживающему несколько доменов (2).

Корневой сервер не знает IP-адреса узла **dc2.loc.comm.org**, но ему известны серверы имен домена **org**, поэтому он отсылает серверу **ns.priv.net** адрес сервера имен **ns.comm.org** (3).

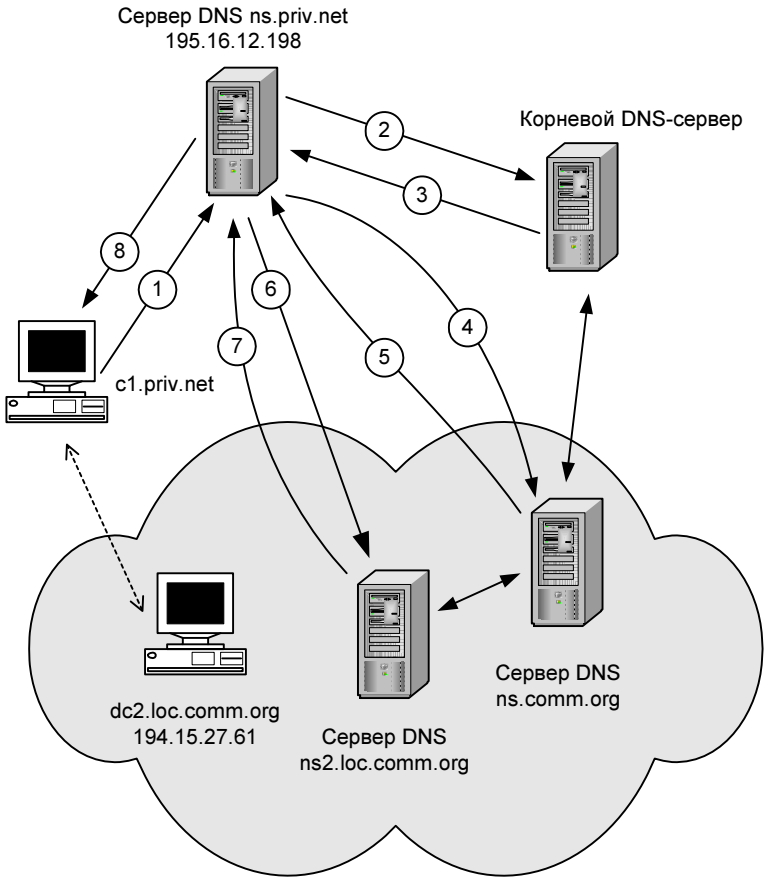


Рис. 8.11. Обработка запроса в DNS

Сервер **ns.priv.net** выполняет запрос к **ns.comm.org** для получения IP-адреса хоста **dc2.loc.comm.ua** (4). Сервер **ns.comm.org** также не знает, как разрешить имя хоста, но ему известен адрес сервера **ns2.loc.comm.org**, который может разрешить проблему, что он и сообщает серверу **ns.priv.net** (5). Сервер имен **ns.priv.net** выполняет запрос к серверу **ns2.loc.comm.org** (6). Сервер **ns2.loc.comm.org** знает IP-адрес хоста **dc2.loc.comm.org** и отсылает эту информацию серверу **ns.priv.net** (7).

На последнем шаге сервер **ns.priv.net** отсылает IP-адрес хоста **dc2.loc.comm.org** (194.15.27.61) хосту **c1.priv.net** (8). По окончании выполнения запроса в кэше сервера **ns.priv.net**, кроме адреса хоста **dc2.loc.comm.org**, будут содержаться адреса серверов, к которым выполнялись обращения. Как легко увидеть из схемы разрешения запроса, сервер **ns.priv.net** является рекурсивным.

Показанная на рис. 8.11 схема обработки запроса DNS очень проста и включает несколько шагов, но, несмотря на это, позволяет продемонстрировать основные принципы функционирования службы имен:

- в случае необходимости разрешения имени или IP-адреса узла хост обращается к своему серверу DNS посредством демона `named` через UDP-порт 53;
- если DNS-сервер в состоянии разрешить запрос, он немедленно отсылает информацию затребовавшему ее хосту, после чего запрос считается выполненным;
- если DNS-сервер не может по каким-либо причинам выдать ответ на поступивший запрос (например, в случае отсутствия необходимых данных в базе и кэше предыдущих запросов), то он обращается к одному из корневых серверов (root servers);
- далее, корневой сервер пытается разрешить запрос и передать информацию затребовавшему ее DNS-серверу зоны, после чего тот отправляет ее клиенту. Если запрос не удается разрешить ни на одном из уровней, то выдается информация об ошибке.

Сервер после передачи данных клиенту кэширует их для дальнейшего возможного использования. Также кэшируются и все дополнительные данные, полученные в процессе обработки запроса.

Служба распознавания имен DNS в операционных системах UNIX реализована посредством открытого пакета программ под названием BIND (Berkeley Internet Name Domain). Существуют версии 4, 8 и 9 этого пакета, причем в большинстве систем в настоящее время используются версии 8 и 9. При анализе функционирования DNS мы будем опираться, в основном, на версии 8 и 9 BIND. В пакет BIND включены следующие основные компоненты: программа-демон `named`, набор библиотечных функций, например, `gethostbyaddr()` и `gethostbyname()`, позволяющих получать информацию из баз данных DNS, а также утилиты `nslookup`, `dig` и `host`, с помощью которых можно выполнить запросы к серверу DNS из командной строки.

Если рассматривать службу распознавания имен в терминологии "клиент-сервер", то работающий демон `named` реализует сервер имен DNS, а набор процедур, выполняющих запросы к `named`, — клиентскую часть и называется распознавателем или резольвером (от англ. *resolve* — решать, разрешать). К резольверу относятся упомянутые нами функции `gethostbyaddr()` и `gethostbyname()`, а также целый ряд других библиотечных функций и, кроме того, утилиты `nslookup`, `dig` и `host`.

Следует сказать, что как и клиентская часть сервиса имен (резольвер), так и серверная (демон `named`) могут устанавливаться на одном и том же хосте.

Как уже упоминалось, демон named принимает запросы на разрешение имен/адресов на UDP-порту 53. Если запросы невозможно разрешить, named переадресовывает их на другие серверы имен, а при получении ответов от них пересылает информацию клиентам и, кроме того, сохраняет полученные данные в кэше. Еще одной важной функцией демона named является копирование данных между серверами зон.

Клиент службы имен

Вначале подробно рассмотрим функционирование клиентской части службы имен. Для функционирования клиента необходимо наличие файла /etc/resolv.conf, содержащего список серверов имен, которым можно посылать запросы. Содержимое этого файла редактируется вручную, если только не используется динамическое присвоение имен службой DHCP, когда файл заполняется автоматически.

В наиболее общем виде формат файла /etc/resolv.conf можно представить таким образом:

```
search имя_домена
nameserver IP-адрес1
nameserver IP-адрес2
. . .
```

Здесь параметр nameserver указывает IP-адрес сервера имен, к которому резольвер выполняет запрос. Максимальное количество серверов, которые могут быть указаны, ограничивается системным параметром MAXNS, причем с каждым ключевым словом nameserver можно указать только один сервер имен. Если в списке указано несколько серверов, то резольвер выполняет запросы к ним в том порядке, в каком они перечислены. В том случае, когда список серверов вообще отсутствует, будет выполнена попытка запроса к серверу имен на данном хосте. Функции резольвера ожидают ответа от сервера в течение определенного интервала времени, по истечении которого запрос будет направлен следующему в списке серверу и т. д.

Параметр search определяет список имен доменов, которые должны просматриваться в поиске серверов имен. Имена доменов в списке должны разделяться пробелами или символами табуляции. Процесс поиска в списке доменов может занимать достаточно длительный период времени и довольно прилично увеличить загрузку сети (сетевой трафик), если обращение происходит к удаленным доменам.

Вот пример записей из файла /etc/resolv.conf:

```
search first.priv.net second.priv.net
```

```
nameserver 172.16.10.45
nameserver 172.16.9.4
```

Клиентские программы могут использовать функцию `gethostbyname()`, которая способна получать информацию как из баз данных DNS, так и из других источников, например, из файлов /etc/hosts. Функция вначале обращается к серверу имен `named`, а в случае неудачи пытается разрешить имя, прочитав файл /etc/hosts. Вот исходный текст простой программы на C++, позволяющей вывести на консоль IP-адрес хоста, доменное имя которого является аргументом программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

int main(int argc, char* argv[])
{
  struct sockaddr_in addr;
  struct hostent *host;
  if (argc != 2)
  {
   printf("Usage: %s hostname\n", argv[0]);
   exit (1);
  }
  host = gethostbyname(argv[1]);
  memcpy(&addr.sin_addr, host->h_addr_list[0], host->h_length);
  printf("%s ==> %s\n", host->h_name, inet_ntoa(addr.sin_addr));
}
```

Преобразование "доменное имя — IP-адрес" выполняется всякий раз при попытке программы установить TCP/IP-соединение с хостом, при этом должно быть известно доменное имя данного узла. Некоторым программам может потребоваться и обратное DNS-преобразование, что можно реализовать при помощи библиотечной функции `gethostbyaddr()`.

Далее приводится исходный текст программы на C, в которой используется функция `gethostbyaddr()`. Программа отображает на экране дисплея имя хоста, IP-адрес которого задается в качестве параметра командной строки:

```
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char* argv[])
{
 struct sockaddr_in local;
 struct hostent *host;
 if (argc != 2)
 {
   printf("Usage:%s IP-address\n", argv[0]);
   exit(1);
 }
 bzero(&local, sizeof(local));
 local.sin_addr.s_addr = inet_addr(argv[1]);
 host = gethostbyaddr(&local.sin_addr.s_addr, 4, AF_INET);
 printf("%s ===> %s\n", argv[1], host->h_name);
 return 0;
}
```

Пример работающей программы (она названа `gethostbyaddr_demo`):

```
# ./gethostbyaddr_demo 194.15.28.10
194.15.28.10 ===> host1
```

В пакет BIND входят утилиты `dig` и `nslookup`, которые разрешают пользователю выполнить запрос DNS в режиме командной строки. Эти программы особенно полезны при отладке службы DNS, а также для получения информации из распределенных баз данных.

Рассмотрим более подробно функционирование одной из них, `nslookup`, которая обычно находится в каталоге /usr/sbin/. Итак, пример использования утилиты:

```
# nslookup www.ibm.com
Server:   yury.ukrtel.net
```

```
Address:  212.16.195.62

Name:     www.ibm.com
Address:  204.146.18.33
```

Здесь для разрешения запроса выполняется обращение к серверу имен **yury.ukrtel.net** (IP-адрес 212.16.195.62), который вернул IP-адрес узла **www.ibm.com**, равный 204.146.18.33. Далее показан пример обратного преобразования:

```
# nslookup 204.146.18.33
Server:    yury.ukrtel.net
Address:  212.16.195.62

Name:     www.ibm.com
Address:  204.146.18.33
```

Команда `nslookup` позволяет напрямую обращаться с запросами к серверам, непосредственно отвечающим за ту или иную зону. Если же ответ поступил от сервера, не отвечающего за зону, для хоста которой запрашивалась информация (например, данные были извлечены из кэша), такой ответ будет помечен как "non-authoritative answer".

Перейдем к анализу функционирования серверной части DNS.

Сервер DNS

Настройка сервера имен представляет собой более сложную процедуру. Напомню, что сервер DNS реализован посредством демона `named` (в операционной системе Solaris — `in.named`) и требует нескольких файлов конфигурации:

- файл named.conf — содержит параметры и указатели на базы данных DNS;
- файл кэша или корневых ссылок (обычно имеет название named.root) — содержит записи о серверах имен корневой зоны;
- файл локальной зоны (обычно называется named.local) — используется для разрешения имени интерфейса обратной связи;
- файл прямой зоны (forward-mapping zone file) — используется для отображения имен хостов в IP-адреса и содержит большую часть информации о зоне. Обычно этот файл имеет описательное имя, указывающее его назначение;
- файл обратной зоны (reverse-mapping zone file) — используется для отображения IP-адресов в имена хостов и ему желательно присвоить какое-либо описательное имя.

Рассмотрим подробно формат этих файлов и начнем с файла named.conf (это имя является стандартным для всех операционных систем). Он может включать следующие директивы:

- `options` — определяет глобальные настройки;
- `server` — определяет параметры удаленного сервера;
- `zone` — указывает зону.

Количество файлов конфигурации сервера имен и их содержимое зависит от назначения данного сервера имен (первичный, вторичный или кэширующий). Например, для кэширующего сервера файл конфигурации named.conf может содержать такие записи:

```
options {
              directory "/var/named";
};
zone "." {
              type hint;
              file "named.root";
};
zone "0.0.127.in-addr.arpa" {
              type master;
              file "named.local";
};
```

Параметры, устанавливаемые директивами `zone`, включаются во все конфигурации серверов имен. Первая директива указывает серверу имен, где искать корневой сервер зоны, что необходимо при запуске демона `named`. Вторая директива `zone` устанавливает сервер первичным по отношению к его собственному адресу обратной связи и хранит эту информацию в файле named.local. Это очень важно, поскольку во многих случаях нужно разрешить собственный адрес обратной связи. Если забыть сконфигурировать зону "0.0.127.in-addr.arpa", то компьютеры сети начнут запрашивать информацию о самих себе у корневых серверов.

Директива `options` определяет каталог по умолчанию, из которого запускается демон `named`, в данном случае это /var/named.

Для первичного сервера имен в домене **myorg.net** файл конфигурации named.conf может выглядеть так:

```
options {
        directory "/var/named";
};
```

```
// a master name server configuration
//
zone "." {
        type hint;
        file "named.root";
};
zone "0.0.127.in-addr.arpa" {
        type master;
        file "named.local";
};
zone "myorg.net" {
        type master;
        file "myorg.net.hosts";
};
zone "15.194.in-addr.arpa" {
        type master;
        file "194.15.rev";
};
```

Опция `directory` указывает на корневой каталог, где хранятся файлы конфигурации. Первые две директивы `zone` имеют тот же смысл, что и в ранее рассмотренном примере, а третья директива указывает на то, что сервер имен является первичным или основным для домена **myorg.net**, информация по которому находится в файле myorg.net.hosts. Четвертая директива `zone` указывает на карту отображения IP-адресов в имена хостов, содержащуюся в файле 194.15.rev.

Далее приводится содержимое файла конфигурации для вторичного сервера имен в домене **myorg.net**:

```
options {
        directory "/var/named";
};
// a slave server configuration
//
zone "." {
        type hint;
        file "named.root";
};
zone "0.0.127.in-addr.arpa" {
        type master;
```

```
        file "named.local";
};
zone "myorg.net" {
        type slave;
        file "myorg.hosts";
        masters { 194.15.28.1; };
};
zone "15.194.in-addr.arpa" {
        type slave;
        file "194.15.rev";
        masters { 194.15.28.1; };
};
```

Отличие данного файла конфигурации от такого же файла для первичного сервера имен в том, что вместо директив `master` здесь указаны директивы `slave`, а также определены ссылки на хост первичного сервера имен (`masters { . . .}`).

Проанализируем теперь назначение и формат записей, используемых в файлах кэша, локальной зоны, файлах прямых и обратных зон:

- SOA (Start of Authority) — обозначает начальную позицию данных в файле зоны и определяет глобальные параметры, действующие в пределах зоны;
- NS (Nameserver) — определяет доменное имя сервера;
- A (Address) — преобразует имя хоста в IP-адрес;
- PTR (Pointer) — преобразует IP-адрес в имя хоста;
- MX (Mail Exchange) — определяет, куда направлять почту для получателя, указанного данным доменным именем;
- CNAME (Canonical Name) — определяет дополнительное имя хоста.

Формат записи для ресурсов DNS выглядит так:

```
[имя]   [ttl]   IN   тип   данные
```

Здесь *имя* обозначает объект домена, `ttl` — интервал времени (в секундах), в течение которого данные должны присутствовать в кэше удаленной машины. Поле `IN` обозначает данную запись как имеющую тип Internet DNS, а поле *тип* принимает одно из значений: SOA, NS, A, PTR, MX, CNAME. Наконец, поле *данные* указывает нужную информацию.

Вот как может выглядеть содержимое файла кэша (он назван named.root):

```
;
.                       3600000  IN  NS FIRST.ROOTSERVERS.NET.
```

```
FIRST.ROOTSERVER.NET.  3600000  IN  A   198.41.0.4
;
.                      3600000      NS SECOND.ROOTSERVERS.NET.
SECOND.ROOTSERVER.NET. 3600000  IN  A   128.9.0.107
;
.                      3600000      NS THIRD.ROOTSERVER.NET.
THIRD.ROOTSERVER.NET.  3600000  IN  A   192.33.4.12
;
```

В этом файле (он еще называется файлом подсказок (hints)) содержатся имена и адреса корневых серверов. В момент загрузки локальный сервер имен по записям этого файла может определить местонахождение корневого сервера для последующей загрузки списка корневых серверов.

Файл локальной зоны (он назван named.local) используется для преобразования адреса 127.0.0.1 (адрес обратной связи) в имя localhost. Формат записей этого файла практически одинаков во всех системах и может выглядеть так:

```
$TTL 86400
@ IN SOA myorg.net. host1.myorg.net.
(
1 ; serial
360000 ; refresh every 100 hours
3600 ; retry after 1 hour
3600000 ; expire after 1000 hours
3600 ; negative cache is 1 hour
)
IN NS myorg.net.
1 IN PTR localhost.myorg.net.
```

Запись SOA указывает зону (**myorg.net**) и почтовый адрес **host1.myorg.net**, по которому можно получить информацию о зоне. PTR-запись отображает адрес 127.0.0.1 на имя **localhost.myorg.net**. Обратите внимание, что доменные имена оканчиваются точкой, а это означает, что добавить дополнительные имена к ним нельзя. Можно использовать этот файл в качестве шаблона (как и все остальные) для установки собственных параметров.

Мы рассмотрели файлы named.conf, named.root и named.local, которые необходимы при настройке вторичных серверов зон и кэширующих серверов. Оставшиеся файлы (прямой и обратной зон) используются в более сложных конфигурациях, которые здесь рассматриваться не будут. Дополнительную информацию по этим файлам и их применению можно найти в Интернете.

8.6.2. Сетевая файловая система NFS

Сетевая файловая система (Network File System, NFS) представляет собой программный интерфейс для доступа к файлам компьютеров, находящихся в сети TCP/IP. Как и в случае DNS, хост может выступать в роли клиента, сервера или клиента и сервера одновременно. Компьютеры, предоставляющие доступ к своим файловым системам по сети, являются серверами NFS, а хосты, получающие доступ к общим сетевым ресурсам, выступают клиентами NFS. Сервис NFS позволяет любому компьютеру в сети получить доступ к файловым системам остальных компьютеров, находящихся в той же сети, обладая в то же время возможностью предоставить доступ к собственным файловым системам.

Доступ к ресурсам сервера клиент может получить только после монтирования файловой системы сервера. После выполнения процедуры монтирования доступ к файловой системе сервера обеспечивается за счет выполнения последовательности удаленных вызовов процедур (Remote Procedure Calls, RPC), в результате чего клиент получает прозрачный доступ к ресурсам сервера. RPC — это программный интерфейс для приложений типа "клиент-сервер", который позволяет выполнить библиотечные процедуры на удаленном сервере. При этом все детали сетевого взаимодействия скрыты в программах клиента и сервера — все сетевые функции инкапсулируются для приложений пакетом RPC и процедурами библиотеки RPC.

С точки зрения пользователя все манипуляции с файлами будут выполняться так, как если бы они находились на локальной машине клиента. Сервис NFS позволяет:

- избегать проблем, связанных с нехваткой дискового пространства на клиентских компьютерах, поскольку отпадает необходимость хранить копии файлов на локальном жестком диске;
- предоставлять клиентские файлы для общего пользования, разместив их на сервере NFS;
- обеспечивать целостность и достоверность информации — все заинтересованные клиенты могут использовать одни и те же файлы, и любые изменения в файлах сразу же становятся доступными для всех;
- упрощать работу клиентов за счет использования хорошо знакомых команд для работы с файлами, например, `cp` вместо `ftp` или `rcp`;
- обеспечивать обмен информацией в гетерогенных (разнородных) сетях.

Доступ клиента к NFS-серверу начинается с процедуры монтирования. В результате этого клиенту возвращается дескриптор файла, с помощью которого

осуществляется дальнейший доступ клиента к файлам в файловой системе сервера. Имена файлов просматриваются на сервере по одному элементу имени за раз, при этом для каждого элемента возвращается новый дескриптор файла. В результате будет получен дескриптор того файла, к которому выполнено обращение. Этот дескриптор будет использоваться при последующих операциях чтения/записи.

NFS выполняет все свои процедуры независимо от количества поступающих запросов так, чтобы клиент мог просто повторно сделать запрос в случае потери соединения. Клиент, например, может выполнять чтение файла в случае отключения или перезагрузки сервера.

NFS-клиенты получают доступ к файлам на NFS-сервере путем отправки RPC-запросов на сервер. В этом случае NFS-клиент реализован как пользовательский процесс, осуществляющий RPC-вызовы на сервере NFS. Сервер NFS реализован как часть операционной системы для повышения эффективности работы.

Рассмотрим основные практические аспекты построения NFS.

Для предоставления файловой системы в общее пользование необходимо ее экспортировать на сервере NFS, добавив записи в файл конфигурации /etc/exports, а также запустить два процесса-демона `rpc.mountd` и `rpc.nfsd`.

Каждая запись в файле /etc/exports определяет экспортируемую файловую систему, а также сетевые хосты, имеющие право доступа к ней. Для файловой системы указывается точка монтирования — каталог, в который она монтируется, затем следует список имен хостов, имеющих к ней доступ. За каждым именем может следовать разделенный запятыми список опций монтирования, взятый в круглые скобки. Например, одному компьютеру можно предоставить доступ только для чтения, другому — для чтения и записи и т. д. Если имена хостов не указаны, то опции монтирования распространяются на всех.

Вот пример содержимого файла /etc/exports:

```
/etc/exports
/pub                (ro,insecure,all_squash)
/home/commres       yuryhost.utel.net(rw)
/mnt/cdrom          education.utel.net(ro)
/home/dataoffice    (noaccess)
```

В этом примере к файловой системе, смонтированной в каталоге /pub (это имя обычно используется для каталога общего доступа), все компьютеры получают доступ только для чтения без проверки прав доступа. Хост **yuryhost.utel.net** получает доступ для чтения и записи к файловой системе,

смонтированной в каталоге /home/commres. Следующая запись предоставляет компьютеру **education.utel.net** доступ к компакт-диску, а последняя запись запрещает всем компьютерам доступ к каталогу /home/dataoffice.

Клиент NFS, прежде чем использовать ресурсы удаленной файловой системы, должен смонтировать ее одним из двух способов:

- сделав соответствующую запись в файле /etc/fstab;
- явно, вызвав команду `mount`.

В строке, описывающей сетевую файловую систему, в поле `filesystemtype` файла /etc/fstab тип файловой системы должен быть определен как NFS. Имя сетевой файловой системы состоит из имени компьютера, на котором она расположена, и путевого имени каталога, в котором она находится. Эти имена разделяются двоеточием. Например, имя host1.priv.net:/home/project относится к файловой системе, смонтированной в каталоге /home/project на компьютере **host1.priv.net**.

Существует несколько специальных опций монтирования NFS, которые можно привести в файле /etc/fstab. Допускается, в частности, указание размера передаваемых и принимаемых дейтаграмм, а также периода, в течение которого компьютер будет ждать ответа от удаленной системы. Можно указать и режим монтирования файловой системы — hard или soft. Если система смонтирована в режиме hard, то в случае, когда удаленная система не отвечает, ваш компьютер будет непрерывно пытаться установить соединение с нею. При монтировании в режиме soft компьютер прекращает попытки и выдает сообщение об ошибке. По умолчанию осуществляется монтирование в режиме hard. Полный перечень опций монтирования NFS со стороны клиента содержится на man-странице, относящейся к команде `mount`. Они отличаются от опций монтирования NFS на стороне сервера.

Файловую систему NFS можно смонтировать явно командой `mount` с опцией `-t nfs`. Например:

```
mount -t nfs -o timeo=20 yuryhost.utel.net:/home/commres
```

Настройка NFS во всех операционных системах выполняется практически по стандартной схеме, хотя есть и некоторые отличия, связанные с реализациями конкретных версий UNIX.

8.6.3. Служба DHCP

Протокол DHCP позволяет клиентам сети получить настройки TCP/IP в момент их загрузки. Так же как и в других сетевых службах, здесь используется механизм "клиент-сервер", при этом сервер DHCP выполняет хранение и управление информацией о сетевой конфигурации клиентов, обеспечивает

их этой информацией по их требованию. Информация, предоставляемая клиентам, включает IP-адрес клиента и информацию о сетевых сервисах, доступных ему.

Основным преимуществом DHCP является способность управлять выделением IP-адресов через механизм лизинга (аренды), что позволяет использовать и освобождать неиспользуемые в течение определенного промежутка времени IP-адреса и выделять их другим клиентам. Это позволяет узлу сети использовать меньший пул (резерв) адресов, чем тот, который задействован для клиентов с постоянными IP-адресами. Кроме того, что очень важно, при перемещении в другую подсеть сервер DHCP присваивает клиенту IP-адрес настройки уже для этой подсети.

Сервис DHCP поддерживает четыре метода выделения IP-адресов для клиентов сети.

- Выделение клиентам постоянных фиксированных адресов. При этом из пула адресов сервера DHCP исключаются постоянно используемые фиксированные адреса, присваиваемые обычно серверам сети.
- Выделение клиентам адресов из пула DHCP с использованием специальных опций, гарантирующих получение клиентами сети определенных адресов во время их загрузки, при этом IP-адреса, выделенные клиентам, не исключаются из общего пула адресов DHCP.
- Автоматическое выделение IP-адресов, во время которого DHCP-сервер выделяет в постоянное пользование IP-адреса из имеющихся в пуле адресов. Эффективность работы DHCP в этом случае снижается, поскольку ограничиваются возможности сервера повторно использовать свободные адреса.
- Динамическое выделение адресов. Сервер DHCP выделяет IP-адрес клиенту для использования в течение определенного интервала времени (лизинг). Клиент может вернуть адрес в любой момент времени, но для продления лизинга должен послать запрос серверу DHCP. Сервер автоматически возвращает IP-адрес клиента в пул адресов, если по истечении времени лизинга клиент не потребовал продления аренды адреса.

В большинстве случаев серверы DHCP используют комбинацию этих методов. Работу службы иллюстрирует рис. 8.12.

Клиент DHCP отправляет широковещательный (broadcast) запрос DHCPDISCOVER (1), содержащий физический адрес сетевого интерфейса (обычно это аппаратный адрес сетевой карты). Запрос отправляется по адресу 255.255.255.255, после чего клиент ожидает ответ от сервера DHCP. При отсутствии ответа в течение определенного интервала времени клиент отправляет запрос повторно.

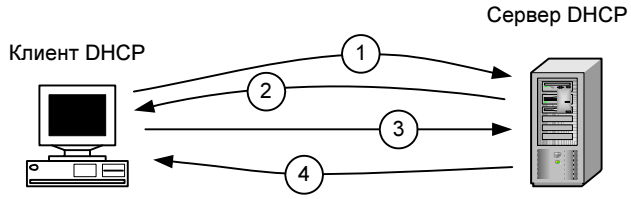


Рис. 8.12. Пример запроса к серверу DHCP: 1 — DHCPDISCOVER; 2 — DHCPOFFER; 3 — DHCPREQUEST; 4 — DHCPACK

Сервер DHCP, получив запрос DHCPDISCOVER, высылает клиенту пакет, содержащий сообщение DHCPOFFER (2). Сервис DHCP использует протокол UDP в качестве транспорта и работает с портами 67 и 68. При этом порт 67 используется сервером, а порт 68 — клиентом DHCP.

Здесь есть один важный момент, о котором следует упомянуть. Как известно, при обмене данными по IP-протоколу программа-сервер всегда использует фиксированный порт, а программа-клиент — произвольный порт, номер которого генерируется сервером в момент соединения.

Клиент DHCP в этом смысле является исключением из общего правила, поскольку при инициализации системы он не имеет своего IP-адреса. Более того, если даже клиент посылает запрос DHCPDISCOVER, сервер не сможет ответить, поскольку в тот момент не знает IP-адреса клиента. Именно поэтому сервер DHCP посылает широковещательный ответ на специально выделенный на всех хостах UDP-порт 68. Ответ читают все хосты, включая клиента, сделавшего запрос. Клиент, прочитав идентификатор транзакции и аппаратный адрес, определяет, что ответ прислан именно ему и продолжает сессию с сервером DHCP.

Полученный клиентом пакет DHCPOFFER содержит данные его сетевой конфигурации. После получения пакета клиент должен ответить серверу в течение 120 секунд, в противном случае сервер DHCP аннулирует предложение DHCPOFFER. Причиной тому является то, что запрос DHCPDISCOVER могут получить несколько серверов, и каждый из них может ответить пакетом DHCPOFFER. Если клиент не подтвердит прием пакета DHCPOFFER, то несколько серверов могут выделить ресурс одному и тому же клиенту одновременно, уменьшив тем самым ресурс IP-адресов для других клиентов.

Если клиент получает несколько пакетов DHCPOFFER, то отвечает только на один, игнорируя все остальные. В качестве ответа серверу высылается пакет DHCPREQUEST (3), содержащий подтверждение данных конфигурации. Сервер DHCP проверяет информацию в пакете DHCPREQUEST, чтобы убе-

диться, что клиент получил правильную информацию, а также наличие предложенных клиенту ресурсов. Если все в порядке, то клиенту отправляется сообщение DHCPACK (4), подтверждающее, что клиент сконфигурирован в соответствии с данными, переданными ему ранее в пакете DHCPOFFER. На этом процедура получения сетевых параметров посредством сервиса DHCP заканчивается.

Существует несколько реализаций DHCP, но мы остановимся на одной из наиболее распространенных — версии, предложенной ISC (Internet Software Consortium). Данная версия используется наиболее популярными операционными системами, такими, например, как Linux, Solaris и FreeBSD. Функции сервера DHCP в данной реализации выполняет демон `dhcpd`.

Установки клиента DHCP обычно не вызывают затруднений. В большинстве операционных систем достаточно, как правило, в сетевых настройках указать на то, что будет использоваться сервис DHCP и IP-адрес самого сервера DHCP.

Демон `dhcpd` считывает необходимые параметры из файла конфигурации /etc/dhcpd.conf, содержащего информацию об обслуживаемых сервером сетях и хостах, а также данные для конфигурирования хостов. Файл dhcpd.conf является обычным текстовым файлом. Вот пример содержимого файла dhcpd.conf:

```
# Define global values that apply to all systems.
default-lease-time 86400;
max-lease-time 604800;
get-lease-hostnames true;
option subnet-mask 255.255.255.0;
option domain-name "priv.net";
option domain-name-servers 172.16.10.9
option interface-mtu 1500;
# Identify the subnet served, the options related
# to the subnet, and the range of addresses that
# are available for dynamic allocation.
subnet 172.16.10.0 netmask 255.255.255.0 {
           option routers 172.16.10.1;
           option broadcast-address 172.16.10.255;
           range 172.16.10.11 172.16.10.29;
}
```

Этот файл определяет очень простую конфигурацию: сервер DHCP обслуживает одну подсеть (172.16.10.0), выделяя динамические IP-адреса клиен-

там данной подсети. Строки, начинающиеся с символа #, являются комментариями.

В первых строках определены общие для обслуживаемой сети и клиентов параметры, причем в первых трех строках установлены параметры для сервера DHCP:

- default-lease-time — определяет время лизинга (аренды) IP-адреса по умолчанию. Время указано в секундах (в данном случае одни сутки);
- max-lease-time — определяет максимальный интервал времени для лизинга (одна неделя в данном случае);
- get-lease-hostnames — требует указывать имя для каждого клиента, которое следует получать от сервиса DNS. Данный параметр может принимать значение `false` (ложь) или `true` (истина). Если он равен `false` (установка по умолчанию), то клиент получает только IP-адрес. Разрешение имени хоста для каждого из динамических IP-адресов замедляет время загрузки клиента сети, поэтому значение `true` обычно присваивают, если в сети используется небольшое число динамических адресов.

В следующих четырех строках описаны опции (ключевое слово option), определяющие общие параметры для клиентов сети. Например, опция `interface-mtu` определяет значение MTU, равное 1500 байтам.

Директивы `subnet` определяют подсети, обслуживаемые демоном `dhcpd`. Тем не менее, в файле конфигурации должны быть описаны сети, к которым сервер DHCP имеет физический доступ, даже в том случае, если в них вообще нет клиентов. Это связано с тем, что демону `dhcpd` требуется информация о сетях для успешного запуска. Параметр `range` устанавливает диапазон динамических адресов для данной подсети (в рассматриваемом примере динамические IP-адреса выделяются из диапазона 172.16.10.11—172.16.10.29).

Посмотрим, какую информацию может отправить демон `dhcpd` клиенту сети 172.16.10.0 при получении от него запроса:

- IP-адрес: 172.16.10.14;
- IP-адрес маршрутизатора по умолчанию: 172.16.10.1;
- IP-адрес широковещательной рассылки: 172.16.10.255;
- маску сети: 255.255.255.0;
- имя домена: priv.net;
- IP-адреса серверов имен: 172.16.10.9;
- MTU для интерфейса Ethernet: 1500.

8.7. Основы программирования сетевых интерфейсов

В этом разделе будут рассмотрены наиболее общие аспекты программирования сетевых приложений в операционных системах UNIX. Материал раздела может быть полезен в первую очередь читателям, желающим познакомиться с основами программирования сетевых приложений, хотя может заинтересовать и профессиональных разработчиков программного обеспечения для UNIX-систем.

Программирование сетевых приложений — это обширная тема и для глубокого ее изучения требуется много времени и усилий. Здесь мы рассмотрим только начальные сведения, которые, тем не менее, помогут при дальнейшем, более глубоком изучении данной темы по другим источникам.

В основе программирования сетевых приложений лежит понятие сокета (гнезда). Сокет представляет собой объект файловой системы, обеспечивающий коммуникацию приложения с сетью. Подобно другим объектам файловой системы UNIX, с сокетом связывается дескриптор файла, что позволяет выполнять операции чтения/записи с ним так же, как и с обычным файлом. Таким образом, при работе с сокетом можно использовать системные вызовы `open()`, `read()`, `write()` и `close()`. Кроме того, для оптимизации операций чтения/записи были разработаны специальные библиотечные функции, такие, например, как `recv()`, `send()` (работают с протоколом TCP), `recvfrom()` и `sendto()` (работают с протоколом UDP).

Сетевые операции выполняются в контексте "клиент-сервер", а это означает, что в любой момент времени один из взаимодействующих процессов является клиентом, а другой — сервером. Как клиент, так и сервер взаимодействуют посредством сокетов, но программные реализации клиента и сервера отличаются. Дальнейший анализ мы будем проводить применительно к стеку протоколов TCP/IP и рассмотрим, как реализуется программный код для клиента и для сервера.

Стек протоколов TCP/IP реализует взаимодействие приложений посредством предварительного установления соединения, что обеспечивает надежную доставку и прием данных. С другой стороны, очень распространенный протокол UDP не требует установления соединения, что упрощает разработку сетевых приложенией с его использованием, но одновременно снижает надежность, поскольку доставка данных адресату не гарантируется.

В общем виде схему взаимодействия клиента и сервера, работающих по протоколу TCP/IP, можно представить следующей схемой (рис. 8.13).

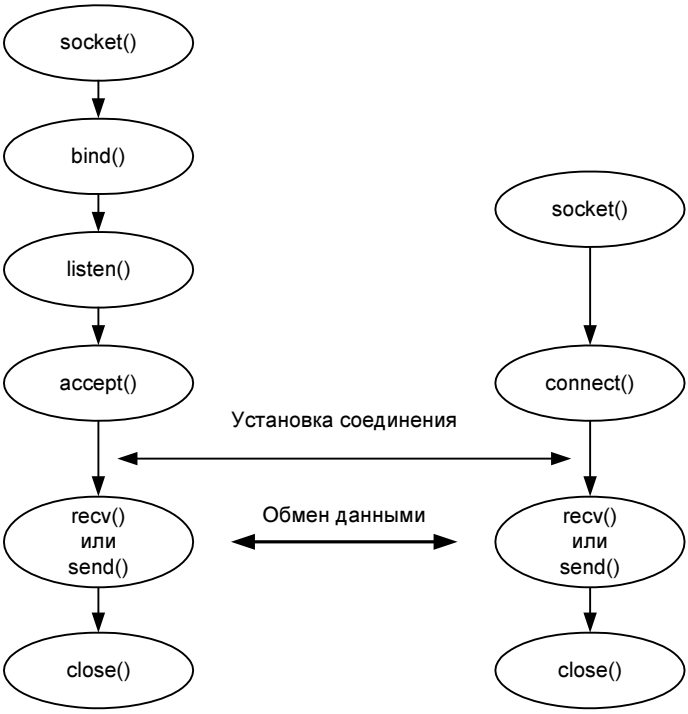


Рис. 8.13. Схема работы приложений по протоколу TCP/IP

Посмотрим, как можно реализовать посредством программного кода схему взаимодействия клиента и сервера, представленную на рис. 8.13. Начнем с сервера.

Первый шаг, который нужно сделать при разработке сервера — создать сокет. Это легко выполнить посредством вызова библиотечной функции `socket()`:

```
int fdsock = socket(family, type, protocol);
```

При вызове функции `socket()` необходимо задать целый ряд параметров, смысл которых таков:

- *`family`* — задает семейство используемых протоколов. Если используется TCP/IP стандарта IPv4 (как в нашем случае), то следует указать `AF_INET`. Константа `AF_INET`, равно как и другие, описана в файле заголовка sys/socket.h;
- *`type`* — задает тип протокола. Для сокетов TCP/IP этот параметр устанавливается как `SOCK_STREAM`. Это означает, что данные передаются и/или

принимаются как непрерывный поток байтов, причем не допускается ни потеря данных, ни их дублирование;

- *protocol* — обычно устанавливается в 0, хотя можно указать одну из констант, например, IPPROTO_IP.

Функция socket() в случае успешного завершения возвращает дескриптор сокета fdsock. В нашем случае для создания потокового TCP/IP-сокета стандарта IPv4 нужно задать функцию socket() со следующими параметрами:

```
int fdsock = socket(AF_INET, SOCK_STREAM, 0);
```

Созданный сокет пока нельзя использовать в операциях обмена данными: удаленный процесс не имеет возможности работать с этим сокетом, поскольку не указан его адрес. Следующий шаг, который необходимо выполнить, — создать локальный адрес и привязать его к сокету, для чего используется функция bind(), имеющая такой синтаксис:

```
bind (fdsock, name, namelen);
```

Здесь *fdsock* — дескриптор сокета, *name* — строка байтов, содержащая IP-адрес и номер порта и которая интерпретируется используемыми протоколами соответствующим образом, а *namelen* — размер данной строки.

Следующий пример демонстрирует привязку локального адреса к сокету:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in server;

fdsock = socket(AF_INET, SOCK_STREAM, 0);
bzero (&server, sizeof (server));
server.sin_family = AF_INET;
server.sin_addr.s_addr = htonl(адрес);
server.sin_port = htons(порт);
bind(fdsock, (struct sockaddr *) &server, sizeof(server));
```

Здесь в качестве второго параметра функции bind() указана структура типа sockaddr_in, поля которой должны быть заполнены соответствующими значениями. Так, поле sin_family должно содержать название семейства протоколов, поле sin_addr.s_addr должно содержать 4-байтовый IP-адрес в сетевом формате, а поле sin_port — значение порта в сетевом формате.

Сетевой формат следования байтов отличается от принятого на локальной машине обратным порядком следования байтов. При выполнении сетевых

операций всегда необходимо преобразовывать представление адреса *адрес* и порта *порт* из локального в сетевой, что и выполняют функции `htonl()` и `htons()`. Первая из них преобразует длинные целые числа, а вторая короткие.

Следующий после привязки адреса шаг — установить только что созданный сокет в режим прослушивания. В этом режиме сокет определяет, какие клиенты требуют соединения с сервером. Установить сокет в режим прослушивания можно при помощи функции `listen()`, имеющей такой синтаксис:

```
listen(fdsock, число_соединений);
```

Здесь *fdsock* — дескриптор сокета, а параметр *число_соединений* определяет длину очереди, т. е. максимальное число одновременно допустимых соединений.

Далее программа-сервер может обрабатывать поступающие соединения, вызывая функцию `accept()`:

```
new_sock = accept(fdsock, (struct sockaddr *) &имя_клиента,
                  sizeof(&имя_клиента);
```

Здесь *fdsock* — дескриптор прослушивающего сокета, *имя_клиента* — байтовая строка, которая будет содержать IP-адрес и порт для вновь созданного клиентского соединения. Наконец, последний параметр определяет адрес имени клиента. При удачном завершении функция `accept()` возвращает дескриптор сокета (положительное число) для только что созданного соединения. Фактически функция `accept()` создает отдельный канал обмена данными, посредством которого и выполняется передача данных между клиентом и сервером.

После этого сервер может принимать данные от клиента или передавать их ему. После окончания обмена данными необходимо закрыть дескриптор рабочего соединения, что выполняет функция `close()`, принимающая в качестве единственного параметра дескриптор сокета.

Сервер должен постоянно обрабатывать входящие запросы, поэтому обработку соединений следует выполнять в бесконечном цикле `while()` или `for()`. Если сервер прекращает работу, то он должен закрыть все открытые дескрипторы сокетов, включая прослушивающий.

Теперь мы можем собрать шаблон нашего приложения-сервера. Вот его примерный исходный текст:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>


#define port MYPORT
```

```
int main(void)
{
 int fdsock, new_sock;
 struct sockaddr_in server, client;

 bzero (&server, sizeof (server));
 server.sin_family = AF_INET;
 server.sin_addr.s_addr = htonl(INADDR_ANY);
 server.sin_port = htons(MYPORT);

 fdsock = socket(AF_INET, SOCK_STREAM, 0);
 bind(fdsock, (struct sockaddr *) &server, sizeof(server));
 listen (fdsock, 5);
 while (1)
 {
   new_sock = accept(fdsock, (struct sockaddr*)&client, sizeof(&client));
   if (new_sock > 0)
   {
     . . .обработка данных . . .
   }
   close(new_sock);
 }
 close(fdsock);
 return 0;
}
```

Этот шаблон программного кода можно использовать для разработки простых серверов TCP. Обратите внимание на строку

```
server.sin_addr.s_addr = htonl(INADDR_ANY);
```

Здесь в качестве параметра функции `htonl` используется `INADDR_ANY`, что означает "любой адрес". Вместо `MYPORT` можно указать любое приемлемое значение для порта из допустимого диапазона. Хочу сказать, что в данном примере для упрощения листинга не анализируются возможные ошибки, но при разработке собственного приложения этот фактор следует учитывать.

Программный код приложения-клиента намного проще (см. рис. 8.13), поскольку клиенту не требуется прослушивать сокет и создавать множественные ветвления, как это необходимо для сервера. Программа-клиент создает сокет при помощи функции `socket()`, после чего устанавливает соединение,

вызвав библиотечную функцию `connect()`. Далее выполняется прием данных от сервера или передача их ему. Функция `connect()` имеет такой синтаксис:

```
connect(fdsock, (struct sockaddr *)&имя_сервера, sizeof(имя_сервера));
```

Здесь параметр `fdsock` является дескриптором сокета, а второй и третий параметры указывают имя, которое должно быть присвоено данному сокету. По завершению обмена данными клиент, так же как сервер, должен закрыть дескриптор сокета.

Вот как выглядит шаблон приложения-клиента:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define port MYPORT
int main(void)
{
 int fdsock;
 struct sockaddr_in server;

 bzero (&server, sizeof (server));
 server.sin_family = AF_INET;
 server.sin_addr.s_addr = htonl(адрес);
 server.sin_port = htons(MYPORT);

 fdsock = socket(AF_INET, SOCK_STREAM, 0);
 connect(fdsock, (struct sockaddr *) &server, sizeof(server));
   . . . обработка данных . . .
 close(fdsock);
 return 0;
}
```

Как видно из листинга, программный код клиента намного проще, чем сервера. Кроме того, следует учитывать, что клиент для соединения использует конкретный IP-адрес, поэтому макрос `INADDR_ANY` здесь использовать нельзя. Одним из вариантов является преобразование имени хоста, на котором работает сервер, в IP-адрес посредством функции `gethostbyname()`.

До сих пор мы не рассматривали функции приема/передачи данных через сетевые соединения TCP. Таких функций две: `recv()` и `send()`. Первая из них

принимает данные от источника, находящегося в сети, а вторая — передает данные сетевому узлу. Синтаксис этих функций представлен далее:

```
send(fdsock, buf, sizeof(buf), flags);
recv(fdsock, buf, sizeof(buf), flags);
```

Здесь первый параметр *fdsock* является дескриптором сокета, второй параметр указывает адрес буфера памяти *buf*, куда должны поступать данные (при приеме) или откуда должны извлекаться данные (при передаче). Третий параметр определяет размер буфера данных, и, наконец, четвертый параметр позволяет установить дополнительные флаги, определяющие способ обработки поступающих или передаваемых данных.

Следует сказать, что вместо функций `recv()` и `send()` можно использовать функции `read()` и `write()`, как и для обычных файловых дескрипторов, хотя функции `recv()` и `send()` являются более предпочтительными, поскольку позволяют посредством дополнительных флагов управлять процессом обмена данными по сети.

Для иллюстрации принципов работы сетевых приложений приведу исходные тексты приложений, одно из которых является сервером TCP, а другое — клиентом. Сервер TCP (его программный файл называется demo_server1) прослушивает порт 7979 и отображает поступающие данные на консоли. Вот исходный текст сервера:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

#define port 7979

int main(void)
{
 struct sockaddr_in server, client;
 int sLocal, sClient;
 char buf[1024];
 int isize = sizeof(client);

 bzero(&server, sizeof(server));
 server.sin_addr.s_addr = htonl(INADDR_ANY);
 server.sin_family = AF_INET;
 server.sin_port = htons(port);
```

```
 sLocal = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
 bind(sLocal, (struct sockaddr*)&server, sizeof(server));
 listen(sLocal, 10);
 printf("TCP Demo Server waits for requests on port %d...\n", port);
 while (1)
 {
 sClient = accept(sLocal, (struct sockaddr*)&client, (socklen_t*)&isize);
 if (sClient)
 {
   int ret = recv(sClient, buf, sizeof(buf), 0);
   if (ret > 0)
   {
     buf[ret] = '\0';
     printf("REQUEST: %s\n", buf);
   }
   close(sClient);
 }
 }
 close(sLocal);
 return 0;
}
```

Программа-клиент отправляет серверу сообщение, текст которого является аргументом командной строки. Исходный текст клиентской программы (она называется `demo_client1`) представлен далее:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#define port 7979

int main(int argc, char* argv[])
{
 struct sockaddr_in client;
 struct hostent *host;
 int sClient;
 int isize = sizeof(client);
```

```
 if (argc != 2)
 {
  printf("Usage: %s message\n", argv[0]);
  exit(1);
 }

 bzero(&client, sizeof(client));
 host = gethostbyname("yuryhost");
 memcpy(&client.sin_addr, host->h_addr_list[0], host->h_length);
 client.sin_family = AF_INET;
 client.sin_port = htons(port);

 sClient = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
 connect(sClient, (struct sockaddr*)&client, sizeof(client));
 send(sClient, argv[1], strlen(argv[1]), 0);
 close(sClient);
 return 0;
}
```

Исходный текст программы несложен для самостоятельного анализа, поэтому остановлюсь лишь на некоторых моментах. Аргументом программы-клиента является либо непрерывная строка, либо строка, содержащая пробелы. Сама строка в этом случае берется в кавычки.

Программа пересылает данные хосту с именем yuryhost на порт 7979. Для получения IP-адреса хоста используется уже знакомая нам функция `gethostbyname()`, возвращающая ссылку на структуру типа `hostent` с заполненными этой функцией полями:

```
host = gethostbyname("yuryhost");
```

Далее полученный IP-адрес клиента копируется в поле `sin_addr` структуры `client` с помощью библиотечной функции `memcpy()`:

```
memcpy(&client.sin_addr, host->h_addr_list[0], host->h_length);
```

Оставшаяся часть программы, думаю, затруднений при анализе не вызовет.

Для компиляции и сборки последних двух приложений можно использовать одну из командных строк:

```
g++ -o имя_программы с-файл
g++ -o имя_программы с-файл -lsocket -lnsl /usr/lib/libsocket.so
g++ -o имя_программы с-файл -lsocket -lnsl /usr/lib/libsocket.so.1
```

Первая из этих командных строк работает в Linux, вторая и третья — в операционных системах Solaris 10 и 11. Желательно проверить, перед тем как выполнять команду, имеются ли в наличии данные библиотеки (`libsocket.so` или `libsocket.so.1`). Следует учитывать и то, что компилятор g++ может располагаться и в каталогах, отличных от тех, что перечислены в переменной `PATH`. В таких случаях можно попытаться обнаружить компилятор C командой `find`, после чего добавить путь поиска в `PATH`, или создать собственный командный файл.

Результат работы программ сервера и клиента вы видите далее:

- сервер:

```
bash-3.00# ./demo_server1
TCP Demo Server waits for requests on port 7979...
REQUEST: Message to be sent to demo_server1
REQUEST: Other message for demo_server1
^C
bash-3.00#
```

- клиент:

```
bash-3.00# ./demo_client1
Usage: ./demo_client1 message
bash-3.00# ./demo_client1 "Message to be sent to demo_server1"
bash-3.00# ./demo_client1 "Other message for demo_server1"
bash-3.00#
```

Работающий процесс-сервер определяется при помощи утилиты `netstat`:

```
bash-3.00# netstat -a|grep IPv4;netstat -a|grep 7979
UDP: IPv4
TCP: IPv4
      *.7979           *.*         0      0 49152      0 LISTEN
bash-3.00#
```

В наших примерах мы использовали так называемые блокирующие сокеты, когда обработка последующего запроса возможна только после окончания обработки текущего. Современные серверные приложения используют иные подходы, позволяющие выполнять так называемую асинхронную обработку данных, когда обрабатывается одновременно много запросов, но данная тема здесь не рассматривается.

Глава 9

Электронная почта

Эта глава посвящена одному из наиболее важных средств коммуникаций с внешним миром — электронной почте. Операционные системы UNIX обладают широкими возможностями по доставке и обработке сообщений электронной почты, как в плане аппаратных конфигураций, так и в выборе программных средств. В этой главе будут рассмотрены программные аспекты функционирования, а также принципы конфигурирования и настройки систем электронной почты.

Большинство программных средств систем электронной почты включает три компонента:

- пользовательский агент (Mail User Agent, MUA), позволяющий читать и отправлять сообщения. Агент читает входящие сообщения, которые приходят на почтовый ящик пользователя, и отправляет исходящие сообщения транспортному агенту для доставки адресатам. Функции пользовательского агента выполняют такие известные программы, как `elm`, `pine`, `mutt`, а самым первым пользовательским агентом была программа `mail`, разработанная корпорацией AT&T. Разработано несколько программ этого класса, включая приложения с графическим интерфейсом. Более того, в настоящее время в таких приложениях широко используется стандарт, позволяющий включать в почтовые сообщения объекты мультимедиа, — так называемый MIME (Multipurpose Internet Mail Extensions, многоцелевые расширения электронной почты для Интернета);
- транспортный агент (Mail Transfer Agent, MTA), который пересылает сообщения с одной машины на другую. В основном транспортный агент работает как "маршрутизатор почты" — он принимает почтовые сообщения, переданные ему или пользовательским агентом (MUA) или другим MTA. Анализируя заголовок почтового сообщения, MTA определяет, какой метод доставки ему использовать, после чего передает сообщение соответствующему

агенту доставки. Кроме того, транспортный агент должен принимать входящую почту от других транспортных агентов. Для работы большинства транспортных агентов необходима поддержка протокола SMTP, который определен в стандарте RFC 821. Для операционных систем UNIX разработано несколько транспортных агентов (`postfix`, `qmail`, `zmailer`, `smail`, `upas` и др.), но самым мощным и распространенным является `sendmail`;

- агент доставки (Mail Delivery Agent, MDA), принимающий сообщения от MTA и выполняющий фактическую их доставку пользователям. К агентам доставки можно отнести программу `procmail`.

В каждом конкретном случае почтовая система может содержать как все компоненты, так и часть из них. Развитые почтовые системы имеют в своем составе все компоненты. Наиболее важным компонентом является MTA, выполняющий основную работу по передаче сообщения. Транспортный агент не выполняет непосредственную доставку сообщения в почтовый ящик, тем не менее, он служит интерфейсом, соединяющим вместе все компоненты почтовой системы. Проиллюстрировать взаимодействие всех компонентов системы электронной почты можно с помощью рис. 9.1.

Предположим, что на хосте **host1.priv.net** работает пользователь alex, имеющий почтовый адрес **alex@host1.priv.net**, и что он отправляет сообщение электронной почты пользователю yury, работающему на хосте **host2.priv.net**. Адрес электронной почты пользователя yury — **yury@host2.priv.net**. Процесс отправки-получения сообщения можно представить в виде последовательности шагов:

1. Пользовательский агент компьютера **host1.priv.net**, в качестве которого может выступать одна из программ `pine`, `elm` и т. д., передает сообщение транспортному агенту, в качестве которого выступает программа `sendmail`, работающая на этом же хосте.
2. Транспортный агент `sendmail` определяет, что почтовое сообщение адресовано пользователю на хосте **host2.priv.net**. Программа `sendmail` сконфигурирована таким образом, что она может доставить сообщение хосту **host2.priv.net** посредством протокола SMTP. Поэтому программа передает сообщение агенту доставки, который встроен в `sendmail` и может работать по протоколу SMTP. Должен заметить, что `sendmail` принадлежит к небольшому числу программ со встроенным MDA, поскольку в большинстве случаев агент доставки реализован в виде отдельной программы.
3. Агент доставки устанавливает соединение с транспортным агентом программы `sendmail`, работающей на хосте **host2.priv.net**, и передает ему сообщение.

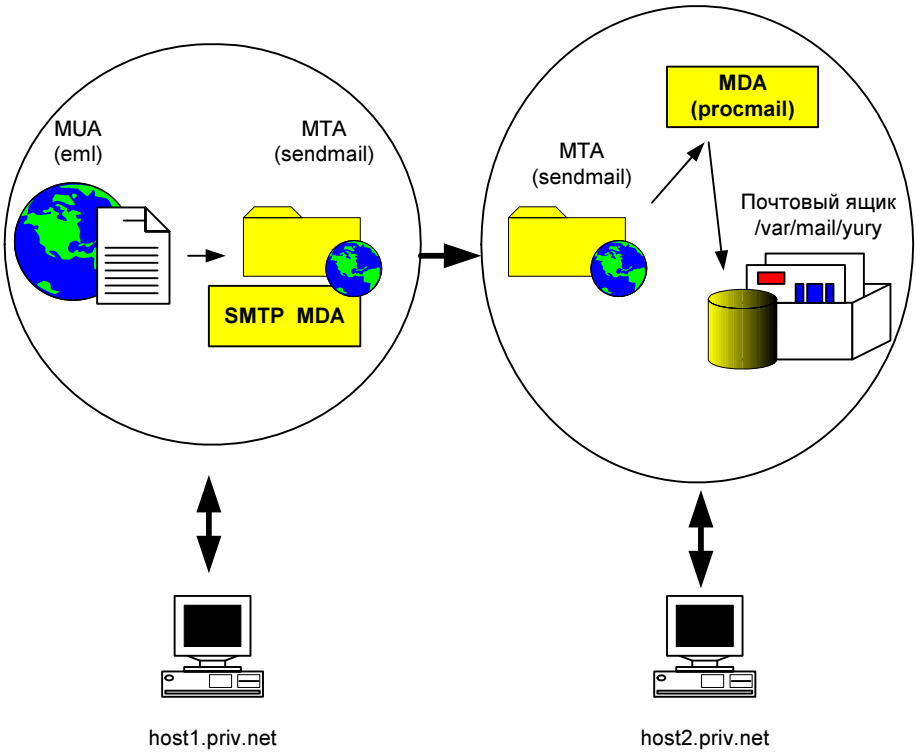


Рис. 9.1. Взаимодействие почтовых систем

4. Транспортный агент на **host2.priv.net** определяет, что сообщение адресовано пользователю на локальном хосте, поэтому передает сообщение агенту доставки, реализованному на этом хосте в виде программы `procmail`.
5. Агент доставки (программа `procmail`) сохраняет сообщение в почтовом ящике пользователя yury (/var/mail/yury).
6. Если пользователь yury вызовет пользовательского агента (например, при входе в систему), то сможет прочитать сообщение.

Файлы почтовых ящиков для операционных систем обычно хранятся в каталоге /var/spool/mail, хотя для получения почты можно выбрать и другие каталоги.

Для приема, отправки и обработки сообщений в большинстве современных почтовых систем используются протоколы SMTP, POP3 и IMAP4. Они определяют процедуры доставки, приема и обработки почтовых сообщений. Например, протокол POP3 предоставляет конечному пользователю доступ к поступившим ему электронным сообщениям. Обычно POP-клиенты при

запросе пользователя на получение почты требуют ввести пароль, что повышает конфиденциальность переписки, при этом запросы программ-клиентов отслеживаются POP-сервером, реализованным как демон `pop3d`. Протоколы электронной почты будут рассмотрены более подробно в последую-щих разделах этой главы.

Аппаратная часть почтовых систем строится с теми или иными вариациями, используя один из двух вариантов. В первом варианте (рис. 9.2) все хосты располагают автономными системами электронной почты.

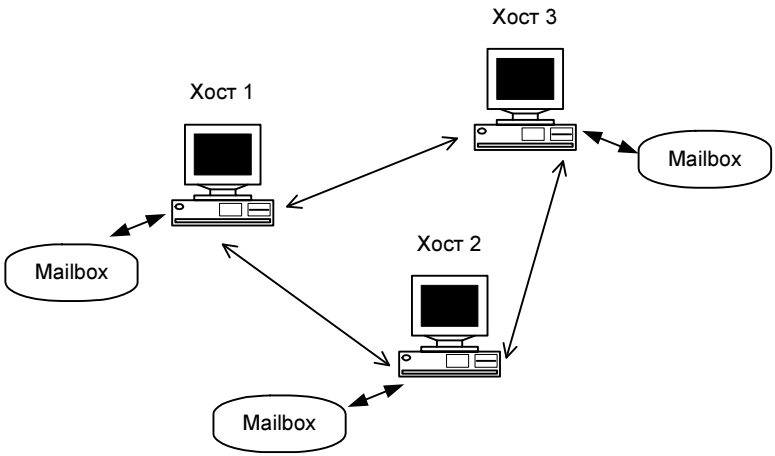


Рис. 9.2. Обмен сообщениями между автономными почтовыми системами

Здесь программное обеспечение каждого из хостов включает необходимый набор агентов (MUA, MTA и MDA), при помощи которых выполняется полная обработка электронной почты. Хорошей иллюстрацией процесса передачи/приема почтовых сообщений от одного хоста к другому может служить рис. 9.1, проанализированный нами ранее. Почтовый ящик (mailbox), а также большинство файлов конфигурации хранятся на локальной машине. Такая аппаратная конфигурация используется в небольших (несколько хостов) сетях. Если требуется передавать сообщения во внешние сети, то программное обеспечение должно обращаться к внешним серверам имен для разрешения почтовых адресов.

Другой тип конфигурации предполагает использование специального сервера электронной почты (рис. 9.3).

Здесь обработку сообщений электронной почты выполняет отдельный сервер электронной почты. Обычно на сервер возлагается ответственность за выполнение централизованных операций, например, управления очередью или

несколькими очередями сообщений, уведомления клиентов о поступивших сообщениях и доставки их на локальные хосты, управления хранилищем сообщений и почтовыми ящиками клиентов. Кроме того, сервер электронной почты берет на себя функцию доставки сообщений в удаленные системы (посредством маршрутизатора электронной почты).

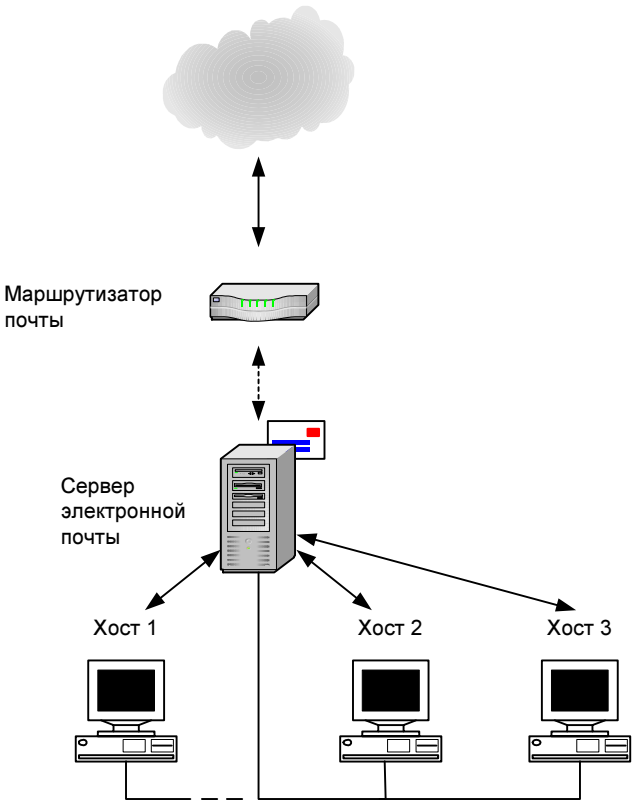


Рис. 9.3. Использование отдельного сервера электронной почты

Серьезным преимуществом применения отдельного сервера электронной почты является то, что он может обеспечить централизованные функции защиты. Программное обеспечение сервера включает, как правило, MTA (обычно программа `sendmail`), а также программы для управления входящими сообщениями (обычно это POP-сервер, и о нем мы поговорим дальше) и преобразованием адресов. На сервере могут работать и другие программы, выполняющие вспомогательные функции. Данная конфигурация почтовой системы используется в средних и больших сетях.

Ознакомимся с методами адресации клиентов электронной почты.

9.1. Адресация электронной почты

Для адресации электронной почты применяются два вида адресов — маршрутно-зависимые и маршрутно-независимые. Первый способ адресации предполагает, что отправителю известны промежуточные хосты, через которые передается сообщение, для того чтобы попасть в пункт назначения. В адресе второго вида просто указывается пункт назначения. В большинстве систем электронной почты используются так называемые интернет-адреса, не зависящие от маршрута и имеющие формат:

пользователь@*хост*

Здесь символ @ отделяет имя пользователя от имени хоста. Вот пример адреса электронной почты:

yury@priv.net

В рассматриваемом примере yury — это идентификатор почтового клиента, а символы, стоящие справа от @, определяют домен, гарантируя однозначное описание клиента. Составные части домена разделяются точками, причем самая правая часть домена может обозначать код страны адресата (например, для Российской Федерации кодом страны является **ru**). Могут использоваться и обозначения сетей (например, для сетей, объединяющих высшие учебные заведения, используются сокращения **edu**, для правительственных учреждений применяется сокращение **gov**, для коммерческих предприятий — **com** и т. д.). Для сетей общего пользования часто выбирается сокращение **net** (как в нашем примере).

Следующий поддомен в нашем примере — priv — однозначно определяется внутри домена верхнего уровня, обозначая либо географический регион, либо какую-нибудь логическую структуру, сгруппированную по определенным признакам (например, структурные подразделения большой организации, город, местность и др.). Аббревиатуры домена второго уровня определяются в соответствии с правилами, принятыми в домене верхнего уровня.

Адреса электронной почты могут включать и больше уровней вложенности, при этом домены нижнего уровня обозначаются в соответствии с правилами, принятыми в доменах верхнего уровня.

Для удобства работы с электронной почтой используются так называемые псевдонимы, позволяющие пользователям переадресовывать почту. Кроме того, псевдонимы применяются в списках рассылки, включающих нескольких получателей, а также для пересылки почты между машинами. Псевдонимы могут использоваться и для того, чтобы обращаться к пользователям по нескольким именам.

Псевдонимы можно указать в файле конфигурации пользовательского агента, в общесистемном файле псевдонимов /etc/aliases (альтернативное название — файл адресных синонимов) или в пользовательском файле пересылки /.forward. Вначале система электронной почты ищет псевдонимы в файле конфигурации пользовательского агента, затем в файле /etc/aliases и, наконец, в пользовательском файле пересылки.

Вот некоторые примеры псевдонимов из файла /etc/aliases:

```
user1: yury
yury: yury@mailserver
```

В первой строке указано, что почту, поступающую на имя `user1`, следует доставлять пользователю yury на локальной машине. Во второй — что всю почту, поступающую на имя yury, следует доставлять на машину `mailserver`.

Помимо списков пользователей, псевдонимы могут применяться для обозначения файла, содержащего список адресов, файла, в который должны добавляться сообщения, а также команды, при помощи которой должны передаваться сообщения.

Для успешной доставки сообщения электронной почты адресату необходимо соблюдать правила оформления электронной почты, определенные международными стандартами (документом RFC 822). Кроме того, электронное письмо должно иметь стандартизованный почтовый электронный адрес.

Сообщение электронной почты кодируется в текстовом файле произвольной формы. При передаче нетекстовых данных (программ, графической информации) применяется перекодировка сообщений, которая выполняется соответствующими программными средствами.

Рассмотрим формат почтового сообщения. Оно состоит из текстовой части, передаваемой адресату, и заголовка, находящегося в начале сообщения. Заголовок отделяется от текста пустой строкой и содержит несколько полей обязательной информации: дату отправления, адрес, обратный адрес, тему сообщения и др. Вот смысл этих полей:

- `Received` — указывает отметку о получении сообщения на хосте;
- `Date` — указывает дату и время отправления письма в стандартном формате;
- `From` — указывает имя отправителя и обратный адрес;
- `Message-Id` — указывает внутренний идентификатор сообщения, присвоенный почтовой службой отправителя. Каждое письмо имеет уникальный идентификатор, который можно использовать для ссылок на письмо, как исходящий номер;

- `To:` — указывает адрес получателя;
- `Subject:` — определяет тему сообщения. Пометка `Re:` означает, что данное сообщение является ответом (от англ. *reply*) на другое сообщение, причем у исходного сообщения и у ответа строка `Subject:` одна и та же.

Для формирования электронного сообщения можно воспользоваться одним из редакторов сообщений для электронной почты.

9.2. Программы *mail* и *mailx*

Самой простой и самой распространенной программой подготовки и отправки почты в операционных системах UNIX является `mail`. Она относится к классу пользовательских агентов и обеспечивает отправку и чтение почты. Данная программа работает даже в тех случаях, когда другие, более современные программы клиентов электронной почты в силу каких-то причин отказываются функционировать. Несмотря на то, что программа `mail` считается устаревшей, имеет смысл изучить принципы ее работы, поскольку функционирование данной программы более понятно и очевидно по сравнению с современными приложениями.

Кроме того, преимуществом программы `mail` является то, что ее удобно использовать в командных файлах, что часто и делается. Далее приводится синтаксис программы:

```
mail [-o] [-s] [-w] [-t] адресат
mail [-e] [-h] [-p] [-q] [-r] [-f файл] [-F адресат ...]
```

Вот смысл некоторых опций программы:

- `-o` — подавляется оптимизация адреса;
- `-s` — символ перевода строки не включается в начало отправляемого сообщения;
- `-w` — при отправке сообщения удаленному пользователю программа не ожидает завершения пересылки;
- `-t` — в сообщение включается строка "`To:` *адресаты*", позволяя тем самым получателю иметь информацию обо всех адресатах сообщения.

Получатель сообщения обычно задается в форме имени пользователя, после чего выполняется отправка почты, если только не указана опция `-F`. Текст отправляемого сообщения читается со стандартного ввода, пока не будет обнаружен символ конца файла (<Ctrl>+<D>), либо не будет введена строка, состоящая из единственной точки. По завершении ввода программа `mail` добавляет сообщение к почтовому файлу каждого из адресатов. В начало сооб-

щения включается заголовок, состоящий из одной или нескольких строк "From ...", за которыми идет пустая строка (если только не была использована опция -s). Если во время ввода нажать клавишу прерывания, то сообщение будет записано в файл dead.letter, что позволяет отредактировать и отправить сохраненное в этом файле сообщение позже.

Если сообщение не было доставлено по каким-либо причинам, то оно возвращается отправителю с указанием информации о причине неудачи.

При приеме почтовых сообщений можно использовать следующие опции:

- -e — устанавливает код завершения без вывода почты (0 означает, что у пользователя имеются входящие сообщения, 1 — их отсутствие);
- -h — позволяет отображать заголовки без вывода текстов самих сообщений;
- -p — позволяет выводить тексты всех сообщений без приглашений;
- -q — позволяет завершить работу команды mail после получения прерывания. В противном случае прерывание вызывает лишь прекращение вывода текста сообщения;
- -r — указывает, что вывод текстов сообщений выполняется в порядке поступления сообщений;
- -f *файл* — в качестве почтового файла используется *файл* вместо принятого по умолчанию;
- -F *адресат* — вызывает переадресацию вновь поступающих сообщений указанному клиенту. Опция допустима только при отсутствии почтовых сообщений у пользователя.

Программа mail выводит тексты сообщений в порядке, обратном их поступлению, т. е. вначале выдаются последние из поступивших сообщений. После обработки последнего сообщения происходит выход из mail или выдается приглашение ?, позволяющее считывать очередную команду со стандартного ввода.

Для обработки и просмотра почты можно использовать такие команды:

- *перевод_строки*, + или n — переход к следующему сообщению;
- d или dp — удалить сообщение и перейти к следующему. Фактическое удаление выполняется только по окончании сеанса работы с mail;
- d n — удалить сообщение с номером n (сообщения нумеруются, начиная с 1, в порядке поступления) и перейти к следующему;
- dq — удалить сообщение и выйти из mail;
- h — отобразить заголовки ближайших к текущему сообщений;
- h n — отобразить заголовок сообщения с номером n;

- `h a` — отобразить заголовки всех сообщений из почтового файла;
- `h d` — отобразить заголовки сообщений, отмеченных к удалению;
- `p` — вновь отобразить содержимое текущего сообщения;
- `-` — отобразить предыдущее сообщение;
- `a` — отобразить сообщение, поступившее во время сеанса работы с `mail`;
- `n` — отобразить сообщение с номером `n`;
- `r [`*адресат ...*`]` — ответить отправителю сообщения, после чего удалить сообщение;
- `s [`*файл ...*`]` — сохранить сообщение в указанных файлах; при этом из почтового файла сообщение удаляется;
- `y [`*файл ...*`]` — см. предыдущую опцию;
- `u [n]` — снять отметку об удалении с сообщения под номером `n` (по умолчанию используется номер последнего прочитанного);
- `w [`*файл ...*`]` — сохранить в указанных файлах только текст (без заголовка сообщения);
- `m [`*адресат ...*`]` — переслать текущее сообщение указанным адресатам;
- `q` или <Ctrl>+<D> — оставить в почтовом файле только неудаленные сообщения и завершить сеанс работы;
- `x` — оставить почтовый файл без изменения и завершить сеанс работы;
- `! `*команда* — выполнить команду интерпретатора shell;
- `?` — отобразить список команд.

О поступлении новых сообщений пользователь уведомляется при входе в систему, а также на протяжении всего сеанса работы. Программа `mail` допускает изменение атрибутов доступа к почтовому файлу, что позволяет обеспечить необходимый уровень защиты. Если режим доступа к почтовому файлу отличается от стандартного, то файл будет сохранен, даже если станет пустым. Если первая строка почтового файла выглядит как

```
Forward to получатель
```

то предназначенная для данного пользователя почта будет пересылаться пользователю *получатель*. При этом к заголовку сообщения добавляется строка "`Forwarded by...`", что позволяет собирать почту на одной машине и получать информацию о пересылке писем. Установка и отмена режима переадресации выполняется посредством опции `-F`:

```
mail -F "список_адресатов"
mail -F ""
```

Первая из команд устанавливает переадресацию на клиентов из *списка_адресатов*, а вторая отменяет переадресацию (список адресатов пуст). Элементы *списка_адресатов* должны разделяться запятыми или пробелами, а сам список должен быть заключен в кавычки. Максимальная длина списка не должна превышать 1024 байта.

Опцию `-s` следует использовать осторожно, т. к. без включения промежуточного символа перевода строки сообщение может быть воспринято как часть заголовка сообщения, что нарушит работу команды `mail`.

Следует заметить, что от системы к системе синтаксис команды `mail` может незначительно меняться, поэтому перед началом работы с этой программой имеет смысл обратиться к соответствующему руководству или к man-страницам.

Еще одной программой для работы с электронной почтой является `mailx` — интерактивная система обработки сообщений. Эта программа помогает выполнять более сложные манипуляции сообщениями по сравнению с `mail`. При чтении почты она позволяет хранить и удалять сообщения, а также отвечать на них. Программа выполняет редактирование и просмотр сообщений, готовых к отправке.

Входящая почта хранится в стандартном для каждого пользователя файле, который называется почтовым ящиком пользователя. После чтения сообщения отмечаются для перемещения во вторичный файл, где они будут сохранены, если только не указано, что данные сообщения более не нужны. Вторичный файл обычно располагается в основном каталоге пользователя $HOME. Сообщения могут быть сохранены и в других файлах, указанных пользователем. Во вторичном файле сообщения остаются до тех пор, пока их не удалят явным образом. Пользователь может получить доступ к вторичному файлу, указав опцию `-f`. Вторичный файл и основной почтовый ящик обрабатываются одними и теми же командами.

Программа имеет следующий синтаксис:

```
mailx [-e] [-f [файл]] [-F] [-h номер] [-H] [-i] [-n] [-N] [-r адрес]
      [-s тема] [-u пользователь] [-U] [адресат ...]
```

В командной строке опции начинаются с символа `-`, а любые другие аргументы задают имена адресатов. Если адресаты не указаны, то `mailx` попытается прочесть сообщения из почтового ящика.

Вот смысл некоторых опций:

- `-e` — проверить наличие почты;
- `-f [файл]` — выполнить чтение сообщений из *файла* вместо почтового ящика;

- `-F` — сохранить сообщение в файле с именем первого адресата;
- `-H` — выводить только заголовки;
- `-i` — игнорировать прерывания;
- `-n` — не выполнять инициализацию с помощью файла mailx.rc (если таковой имеется);
- `-N` — не выводить список заголовков сообщений;
- `-r` *адрес* — передавать *адрес* сетевому программному обеспечению доставки почты, при этом запрещены все команды с тильдой;
- `-s` *тема* — устанавливать *тему* в поле заголовка `"Subject"`;
- `-u` *пользователь* — прочитать почтовый ящик пользователя. Это возможно только в том случае, если разрешено чтение почтового ящика *пользователя*.

Утилита `mailx` запускается в режиме командной строки. В момент запуска на экран выводятся заголовки первых сообщений и приглашение к вводу стандартных команд. При отправке почты `mailx` также ожидает ввода данных.

Если в командной строке не указана тема, то выдается приглашение на ее ввод. После того как сообщение набрано, программа `mailx` читает его и сохраняет во временном файле. Команды можно вводить с помощью указания тильды `~` в качестве первого символа строки, затем первого символа команды и необязательных аргументов.

В любой момент времени поведением утилиты `mailx` управляет множество переменных окружения, которые делятся на две группы: флаги и переменные со значениями. Они создаются и удаляются с помощью команд `set` и `unset`.

В командной строке адресаты могут задаваться тремя способами: как входные имена пользователей, как команды shell или как псевдогруппы. В качестве входных имен могут выступать произвольные сетевые адреса. Если оказалось, что почту невозможно доставить, то `mailx` пытается вернуть ее в почтовый ящик отправителя. Если имя получателя начинается с символа программного канала `|`, то остальная часть имени рассматривается как команда shell, выступающая в роли фильтра. Такой подход предоставляет интерфейс с любой программой, принимающей данные со стандартного ввода. Псевдогруппы создаются с помощью команды `alias` и являются списком адресатов произвольного типа.

В режиме чтения почтовых сообщений программа `mailx` использует те же команды, что и `mail`. После обработки последнего сообщения отображается приглашение `?`, позволяющее считывать очередную команду со стандартного ввода.

Наш анализ работы программ `mail` и `mailx` закончим примером сеанса работы с `mailx`. Предположим, что пользователь yury отправил два сообщения пользователю root:

```
$ bash
bash-3.00$ mailx root
Subject: TEST MESSAGE 1
STRING1 from user yury

EOT
bash-3.00$ mailx root
Subject: TEST MESSAGE 2
string2 from yury

EOT
bash-3.00$
```

Хочу напомнить, что текст сообщения следует заканчивать символом точки в первой колонке следующей строки.

Пользователь root читает пересланные ему сообщения:

```
bash-3.00# mail
From yury@localhost Mon Mar 13 15:41:58 2006
Date: Mon, 13 Mar 2006 15:41:58 GMT
From: yury@localhost
Message-Id: <200603131541.k2DFfwoQ000995@localhost>
To: root@localhost
Subject: TEST MESSAGE 2
Content-Length: 18

string2 from yury

? n
From yury@localhost Mon Mar 13 15:41:16 2006
Date: Mon, 13 Mar 2006 15:41:15 GMT
From: yury@localhost
Message-Id: <200603131541.k2DFfFwk000991@localhost>
To: root@localhost
Subject: TEST MESSAGE 1
```

```
Content-Length: 23

STRING1 from user yury

?
```

Обратите внимание на порядок просмотра сообщений: при запуске `mailx` первым будет прочитано последнее пришедшее сообщение, вторым — предпоследнее и т. д. Команда `n` позволяет прочитать следующее, т. е. более раннее по дате сообщение, что видно из листинга. Нумерация сообщений ведется от предыдущих сообщений к последующим, т. е. если в почтовом ящике пользователя находятся два сообщения, то следующее поступившее сообщение будет иметь номер 3. В этом можно убедиться, набрав следующие команды:

```
? #
Current message number is 2
? -
From yury@localhost Mon Mar 13 15:41:58 2006
Date: Mon, 13 Mar 2006 15:41:58 GMT
From: yury@localhost
Message-Id: <200603131541.k2DFfwoQ000995@localhost>
To: root@localhost
Subject: TEST MESSAGE 2
Content-Length: 18

string2 from yury

? #
Current message number is 3
```

Здесь первая команда `#` отображает номер текущего сообщения (он равен 2), затем команда `-` извлекает предыдущее (более позднее) сообщение (его номер равен 3).

Просмотреть заголовки всех сообщений можно с помощью команды `h a`:

```
? h a
3 letters found in /var/mail/root, 0 scheduled for deletion, 0 newly arrived
>   3     563    yury@localhost  Mon Mar 13 15:41:58 2006
    2     568    yury@localhost  Mon Mar 13 15:41:16 2006
    1     574    root@localhost  Mon Mar 13 15:26:31 2006
```

Для удаления сообщения с определенным номером следует выполнить команду

```
d номер_сообщения
```

Например, для удаления сообщения 2 выполним команду `d 2` и проверим список сообщений:

```
? d 2
? h a
3 letters found in /var/mail/root, 1 scheduled for deletion, 0 newly arrived
    3     563    yury@localhost  Mon Mar 13 15:41:58 2006
>   2  d  568    yury@localhost  Mon Mar 13 15:41:16 2006
    1     574    root@localhost  Mon Mar 13 15:26:31 2006
? q
```

После выполнения этих команд видно, что сообщение 2 помечено для удаления, но фактическое удаление сообщения произойдет после завершения программы `mailx` (команда `q`).

Более подробную информацию о командных опциях утилиты `mailx` можно найти в справочной системе UNIX.

9.3. Программа *sendmail*

Одним из основных и наиболее распространенных средств рассылки почты в Интернете является программа `sendmail`. В данном разделе мы проведем детальный анализ работы программы. Напомню, что `sendmail` по своим функциональным характеристикам является транспортным агентом, выполняющим функции интерфейса между пользовательскими агентами и агентами доставки. Более того, при обмене почтовыми сообщениями через Интернет программа сама является агентом доставки.

Программа `sendmail` позволяет выполнять:

- управление сообщениями, созданными пользователями;
- анализ адресов получателей сообщений;
- выбор соответствующего транспортного агента или агента доставки;
- преобразование адресов в форму, понятную агенту доставки;
- переформатирование заголовков, если это необходимо;
- передачу преобразованного сообщения агенту доставки.

Кроме того, программа `sendmail` генерирует сообщения об ошибках и возвращает отправителю сообщения, которые не могут быть доставлены.

Программа sendmail тесно связана с программами подготовки и просмотра почтовых сообщений и позволяет организовать почтовую службу локальной сети таким образом, чтобы можно было обмениваться почтой с другими серверами почтовых служб посредством специальных шлюзов. Наконец, программу sendmail можно настроить для работы с различными почтовыми протоколами, такими, например, как UUCP и SMTP.

Кроме основных функций, перечисленных выше, утилиту sendmail можно сконфигурировать для поддержки:

- списка адресов-синонимов;
- списка адресов рассылки пользователя;
- автоматической рассылки почты через шлюзы;
- очередей сообщений для повторной рассылки почты в случае отказов при рассылке;
- функционирования в качестве SMTP-сервера;
- доступа к адресам машин через сервис имен DNS;
- доступа к внешним серверам имен.

Универсальность и мощь программы sendmail проявляются и в том, что она может выполнять разнообразные функции, являясь вспомогательной для других почтовых программ. Например, sendmail может использоваться программой подготовки сообщений для отправки уже созданных сообщений, программой получения почты для пересылки полученной почты или самим пользователем для отправки по почте файла или сообщения.

Рассмотрим, как выполняется доставка сообщений программой sendmail. Процесс отправки электронной почты условно можно разбить на два этапа: создание очередей почтовых сообщений и собственно отправку. Лучше всего этот процесс иллюстрирует рис. 9.4.

Проанализируем схему отправки сообщений, представленную на рис. 9.4, более подробно. Программа sendmail обычно работает в фоновом режиме, ожидая новых сообщений, которые могут поступать из командной строки, устройства стандартного ввода, посредством SMTP-протокола или из очереди сообщений.

При получении сообщений из командной строки или стандартного ввода программа sendmail должна в качестве параметра получить адрес доставки сообщения. Далее выполняются такие действия:

- определяется адрес отправителя;
- из командной строки выбирается адрес получателя, причем оба адреса преобразуются в соответствии с описанием файла конфигурации;

- определяется способ доставки сообщения, а заголовок размещается в оперативной памяти для последующих преобразований.

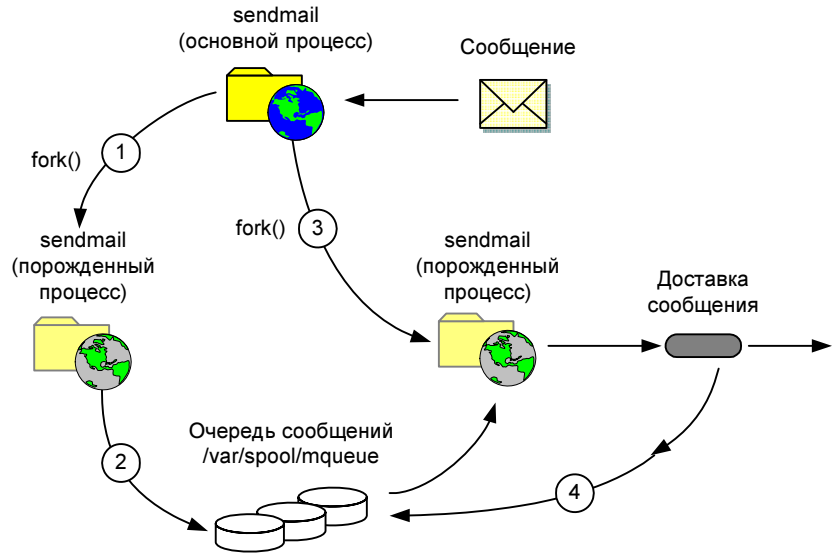


Рис. 9.4. Схема передачи сообщений при помощи `sendmail`

При поступлении очередного сообщения `sendmail` порождает новый процесс для его обработки с помощью системного вызова `fork()` (1). Основной процесс, передав обработку сообщения порожденному процессу, продолжает прослушивать соединение в ожидании следующих сообщений.

Порожденный процесс добавляет поступившее сообщение в очередь для отправки (обычно это /var/spool/mqueue), из которой сообщение будет отправлено адресату (2). Почтовое сообщение, поступившее в очередь, преобразовывается следующим образом:

- адреса отправителя и получателя преобразуются в сетевой формат получателя почты;
- при необходимости в заголовок сообщения добавляются строки, позволяющие получателю отвечать на принятое сообщение.

Если данное сообщение можно доставить клиенту немедленно, то выполняется его пересылка, при этом сообщение из очереди удаляется.

Доставкой сообщений из очереди занимается другой порожденный процесс (3). Если сообщение по каким-либо причинам нельзя доставить адресату, то оно

возвращается в очередь сообщений (4), после чего через определенный интервал времени (определяется при настройке) будет выполнена повторная попытка доставить его адресату. Основной процесс периодически создает новый процесс для проверки очереди сообщений и отправки этих сообщений клиентам.

Если сообщение не может быть доставлено, оно либо возвращается отправителю с диагностическим сообщением, указывающим на причину сбоя, либо сохраняется в файле dead.letter в домашнем каталоге отправителя.

Программа `sendmail` оперирует сообщениями, состоящими из трех частей: конверта, заголовка и тела сообщения. Конверт состоит из адреса отправителя, адреса получателя и информации рассылки, которая предназначена для программ подготовки, рассылки и получения почты. Отправитель и получатель почтового сообщения не могут прочитать содержимое конверта.

Заголовок состоит из стандартных текстовых строк, в которых содержатся адреса, информация о рассылке и данные. Обычно заголовок является частью подготовленного пользователем текстового файла, хотя может быть создан и добавлен к телу сообщения программой подготовки почты. Данные из заголовка могут быть использованы для оформления конверта сообщения.

Программа `sendmail` использует протокол SMTP, причем может функционировать и как клиент, и как сервер. Напомню, что протокол SMTP в настоящее время является стандартом для приложений, занимающихся обработкой электронной почты в Интернете. Если `sendmail` выполняет функции сервера SMTP, то он работает как процесс-демон, прослушивая TCP-порт 25 и, в случае получения сообщения, устанавливает соединение с удаленным клиентом SMTP. Как правило, таким клиентом является другая программа `sendmail`.

Программа подготовки почты на локальной машине также может использовать SMTP, при этом `sendmail` открывает программный канал для межпроцессного обмена.

Отправка сообщений электронной почты может выполняться как на удаленный хост, так и на локальный (местный) компьютер. В случае оправки сообщения на удаленный хост программа `sendmail` открывает программный канал и запускает программу рассылки, командная строка которой описана в файле конфигурации. В этом случае `sendmail` записывает заголовок и тело сообщения в программный канал.

Если программа рассылки использует протокол, отличный от SMTP, то адрес получателя передается через этот программный канал. В случае использования SMTP открывается двунаправленный канал для интерактивного взаимодействия с удаленным сервером SMTP. Наконец, возможен вариант, когда удаленный сервер в качестве транспортного протокола использует TCP (что

чаще всего и бывает), тогда `sendmail` вообще не вызывает внешнюю программу рассылки, самостоятельно инициируя TCP-соединение с удаленным сервером SMTP.

Доставка локальной почты выполняется по-другому. Если `sendmail` определяет, что адреса доставки являются местными, то она обращается к файлу адресных синонимов /etc/aliases, выполняя преобразование (расширение) адресов. Файл адресных синонимов можно использовать для перенаправления почты в файлы или для обработки локальными программами. Вот пример записей файла /etc/aliases:

```
# RFC 822 requires that every host have a mail address "postmaster"
postmaster      : root
```

Эта запись указывает на то, что любая почта, предназначенная клиенту postmaster, должна направляться суперпользователю root.

Следующая запись перенаправляет почтовые сообщения в "никуда":

```
# Aliases to handle mail to msgs and news
nobody          : /dev/null
```

поскольку данные, выводимые в устройство /dev/null, теряются.

В записях можно указывать конвейер команд, как в этом случае:

```
# an automatic bug-registering database
program-bugs: |/usr/local/bin/program-bug-tracker
```

Можно перенаправить почту сразу нескольким клиентам, адреса которых разделяются запятыми:

```
# a makeshift mailing list
project-list:
alex@host1.priv.net,yury@host2.priv.net,someone@host3.priv.net
```

Здесь почтовое сообщение для клиента project-list будет перенаправлено по всем перечисленным адресам.

Программа `sendmail` не может напрямую прочитать содержимое файла /etc/aliases. Вместо этого данный файл конвертируется в файл базы данных (DB или DBM) командой `newaliases`, после чего он уже может использоваться программой `sendmail`. При каждом изменении файла /etc/aliases необходимо запускать команду `newaliases` для обновления баз данных. Следует отметить, что, кроме стандартного, пользователь может создать собственный файл адресных синонимов для управления рассылкой персональной почты.

Для разрешения адресов программа `sendmail`, кроме файла /etc/aliases, может обращаться и к различным сервисам службы имен, используя, например,

файл /etc/nsswitch.conf. Если в домашнем каталоге пользователя существует файл .forward, то он также просматривается в поиске возможных адресатов для пересылки. После преобразования адресов почта отправляется программе местной рассылки.

Каким же образом программа `sendmail` распознает тип адреса (локальный, UUCP или SMTP)?

Замечу, что анализ типа адресов является необходимым и важным этапом при функционировании `sendmail`, поскольку тип адреса получателя указывает программе на способ рассылки сообщения. В этом процессе участвуют, как минимум, два сервиса: система рассылки почтовых сообщений и DNS. Кроме того, возможность вызова программы доставки включена в правила преобразования адресов отправителя и получателя.

Если система определяет, что дальнейшее преобразование адреса нецелесообразно, то начинает выполняться программа доставки. Следует отметить, что бóльшая часть сообщений об ошибках при рассылке сообщений связана с определением адреса получателя.

Определяющим моментом при анализе адреса является форма представления различных типов почтовых адресов.

Местные адреса можно представить в форме:

- ***user***
- ***user*@localhost**
- ***user*@localhost.localdomain**
- ***user*@alias**
- ***user*@alias.localdomain**
- ***user*@[local.host.internet.address]**
- **localhost!*user***
- **localhost!localhost!*user***
- ***user*@localhost.uucp**

Адреса UUCP можно представить так:

- **host!user**
- **host!host!user**
- ***user*@host.uucp**

В практическом плане передача сообщений при использовании протокола UUCP осуществляется от машины к машине по указанному адресу пути. Если машина, с которой отправляется почта, непосредственно связана по

протоколу UUCP со следующей машиной, указанной в адресе, то почта передается на эту машину. Если такое соединение отсутствует, то рассылка почты не выполняется, и выдается сообщение об ошибке. Файл конфигурации sendmail.cf должен содержать детальное описание маршрутов для пересылки сообщений по протоколу UUCP. Если сообщение ушло с машины отправителя, то это еще не означает, что его получит адресат. Промежуточный узел может вернуть почту отправителю, если не сможет ее разослать.

Адреса SMTP — это стандартные адреса Интернета, описанные в документе RFC 822, например:

- ***user@host***
- ***user@host.domain***
- ***@host1,@host2,@host3:user@host4***
- **user*@remote_hosts_internet_address***

Почтовые сообщения с адресами SMTP рассылаются с использованием протокола SMTP. В отличие от UUCP, протокол SMTP работает по-другому: почта отправляется только в том случае, если установлено соединение с сервером, получающим сообщение. В этом случае передача почты синхронизируется командами клиента и сервера SMTP в режиме on-line.

Если операционная система использует службу имен DNS, то команда `sendmail` может определять адреса получателей посредством этой службы. В противном случае `sendmail` попытается определить адреса самостоятельно.

Можно использовать и вариант смешанной адресации, например:

- ***user%hostA@hostB*** — сообщение отправляется с hostB на hostA;
- ***user!hostA@hostB*** — сообщение отправляется с hostB на hostA;
- ***hostA!user%hostB*** — сообщение отправляется с hostA на hostB.

В режиме смешанной адресации доставка почты выполняется по смешанному сценарию. В любом случае информацию о доставке и маршрутизации сообщения можно определить из заголовка полученного сообщения.

Настройка программы `sendmail` выполняется посредством файла конфигурации sendmail.cf. Содержимое этого файла состоит из нескольких частей или секций:

- секция описания характеристик данного хоста, в которой указываются его имя и другие характеристики;
- секция описания макроопределений `sendmail`, необходимых для функционирования в локальной сети (имя домена, имя хоста и др.);

- ❒ секция описания классов (групп имен), используемых программой для рассылки почты, например, других почтовых служб;
- ❒ номер версии файла конфигурации, который должен изменяться каждый раз при внесении в файл изменений;
- ❒ внутренние макроопределения `sendmail` — переменные, использующиеся при взаимодействии с другими транспортными агентами;
- ❒ опции команды `sendmail`, определяющие режимы работы программы. Опции можно также задавать и в виде параметров командной строки;
- ❒ секция порядка сообщений программы `sendmail` (precedence);
- ❒ секция доверенных пользователей — здесь указывается, кому разрешено переопределять адресата отправки;
- ❒ секция описания формата заголовка почтового сообщения, в которой определяются поля, отображаемые в заголовке, а также их формат. Многие из них создаются программой `sendmail` на основе информации, содержащейся в конверте почтового сообщения;
- ❒ секция правил преобразования адресов, используемых программой для выбора маршрута рассылки почтовых сообщений;
- ❒ описание программ рассылки (путевых имен и параметров командной строки). Обычно это программы местной, UUCP- и SMTP-рассылки;
- ❒ набор правил преобразования адресов, которые являются инвариантными (неизменными) по отношению к конфигурации хоста;
- ❒ аппаратно-зависимая часть общего правила преобразования адресов. Здесь определяется способ рассылки почты, при этом секция для SMTP отличается от секции для UUCP.

В большинстве случаев все изменения, которые вносятся в файл конфигурации, связаны только с именем хоста, домена и шлюзов к другим почтовым службам. С помощью указанной в секциях информации решаются три основные задачи:

- ❒ определение окружения `sendmail`;
- ❒ анализ и преобразование адресов электронной почты;
- ❒ рассылка сообщений при помощи программ рассылки.

Редактируя файл конфигурации, следует учитывать ряд правил:

- ❒ вся информация локального характера записывается в начале файла;
- ❒ команды одного типа собраны в компактные группы;
- ❒ бóльшую часть записей файла составляют правила преобразования адресов;
- ❒ в конце файла описаны программы рассылки электронной почты.

Мы не будем подробно останавливаться на формате файла конфигурации sendmail.cf программы `sendmail`, поскольку это сложная тема и ей посвящены целые книги. Примеры настройки программы `sendmail` посредством файла конфигурации имеются в Интернете или в специальной литературе.

Рассмотрим способы запуска программы `sendmail`. Программа `sendmail` может запускаться из командной строки без параметров или с параметрами. Вот пример использования программы `sendmail`:

```
# sendmail yury
Hello from sendmail running under root
.
#
```

Единственным параметром здесь является имя адресата. Пользователь yury получит следующее сообщение:

```
$ mail
Mail version 8.1 6/6/93.  Type ? for help.
"/var/spool/mail/yury": 1 message 1 new
>N  1 root@localhost.local  Wed Mar 15 14:00  14/623
& p
Message 1:
From root@localhost.localdomain  Wed Mar 15 14:00:42 2006
Date: Wed, 15 Mar 2006 14:00:17 +0200
From: root <root@localhost.localdomain>

HELLO from sendmail running under root

& q
Saved 1 message in mbox
```

Рассмотрим некоторые параметры, наиболее часто используемые с программой `sendmail`:

- `-ba` — устанавливает режим ввода, при котором завершение ввода определяется комбинацией CRLF (Carriage Return — Line Feed);
- `-bd` (`bD`) — программа `sendmail` запускается как процесс-демон в фоновом режиме, ожидая поступления запросов на подключение;
- `-bi` — программа выполняет инициализацию базы данных aliases (DB- или DBM-файл, полученный в результате преобразования файла /etc/aliases). Для успешного выполнения данной операции суперпользователь root должен быть владельцем файла базы данных и иметь права записи в данный файл;

- `-bm` — доставка почты выполняется обычным способом (установка по умолчанию);
- `-bp` — программа выводит на консоль информацию об очереди сообщений;
- `-bs` — программа использует протокол SMTP, как описано в стандарте RFC 2821 SMTP;
- `-bv` — программа проверяет имена, не выполняя доставки сообщений или помещения их в очередь. Этот режим позволяет проверить списки пользователей электронной почты;
- `-C file` — предписывает программе `sendmail` использовать альтернативный файл конфигурации вместо файла sendmail.cf, принятого по умолчанию;
- `-h N` — устанавливает количество переходов (hops), равное `N`. Счетчик переходов увеличивается каждый раз при повторной обработке сообщения. При достижении значения `N` генерируется сообщение об ошибке. Чаще всего подобная ситуация возникает при закольцовывании сообщения вследствие ошибок в файлах aliases и .forward;
- `-n` — запрещает программе использовать адресные синонимы;
- `-q[time]` — предписывает программе `sendmail` обрабатывать сохраненные в очереди сообщения через указанный интервал времени `time`. Если параметр `time` отсутствует, очередь обрабатывается только один раз. Параметр `time` представляет собой число, после которого указывается единица измерения (`s` — секунды, `m` — минуты, `h` — часы, `d` — дни, `w` — недели). Например, `-q1h30m` означает, что тайм-аут будет равен 1 часу 30 минутам. Альтернативный вариант записи — `-q90m`;
- `-qp[`*time*`]` — опция схожа с `-q[`*time*`]`, за исключением того, что вместо периодического вызова функции `fork()`, создающей дочерние процессы для обработки очереди сообщений, программа `sendmail` порождает единственный дочерний процесс для обработки очереди сообщений.

Программу `sendmail` можно запускать как процесс-демон при начальной загрузке системы, например:

```
# /usr/lib/sendmail -bd -q30m
```

Здесь опция `-q` определяет, как часто следует обрабатывать очередь сообщений (в данном случае каждые 30 минут). Этот интервал времени выбран произвольно, хотя при интенсивном обмене сообщениями его следует выбрать в пределах 10—15 минут, что для большинства систем является оптимальным. Если выбрать значение этого параметра слишком ма́лым, то могут возникнуть проблемы с производительностью, особенно при резком возрастании количества сообщений в очереди, например, при медленной сети.

Опция `-bd` указывает на то, что программа `sendmail` должна выполняться как демон и прослушивать TCP-порт 25 для приема входящей почты.

Следует сказать, что исполняемый файл `sendmail` может находиться в другом каталоге, отличном от /usr/lib (например, в операционной системе Red Hat Linux программа `sendmail` расположена в каталоге /usr/sbin).

Во всех операционных системах предусмотрен вариант запуска программы `sendmail` с помощью командных скриптов во время загрузки системы. В Solaris, например, используется скрипт /etc/init.d/sendmail, а в Red Hat 9 запускается командный скрипт /etc/rc.d/init.d/S80sendmail.

Рассмотрим некоторые важные аспекты практического применения программы `sendmail` для передачи электронной почты. В большинстве случаев программа запускается как процесс-демон, работающий в фоновом режиме и обслуживающий поступающие запросы на передачу сообщений.

При запуске программа `sendmail` допускает использование различных комбинаций параметров командной строки, мы же рассмотрим только некоторые из них. Во многих случаях в командной строке для `sendmail` задаются два параметра: `-bd`, указывающий на то, что программа будет выполняться как процесс-демон, и `-q`, определяющий периодичность обработки очереди сообщений.

Иногда интерпретатор shell не находит исполняемый файл программы `sendmail`, и требуется определить, установлена ли эта программа в операционной системе. Программа `sendmail` является классической для UNIX и, как правило, включается во все дистрибутивы операционной системы, тем не менее, проверить ее наличие на диске можно при помощи команды

```
find / -name "sendmail" -print
```

Запустим программу `sendmail` и детально проанализируем, какие процессы происходят во время ее работы. Для запуска воспользуемся командной строкой

```
# sendmail -bd -q30s
```

В данном случае `sendmail` запускается как процесс-демон, который каждые 30 секунд будет проверять очередь сообщений для отправки их адресату или следующему узлу маршрута электронной почты.

После запуска программы `sendmail` проверим ее наличие в таблице процессов. Все дальнейшие рассуждения будем проводить относительно функционирования `sendmail` в операционной системе Linux, хотя для других UNIX-систем все выкладки будут отличаться незначительно от приведенных далее. Весьма полезным в этом случае может оказаться и рис. 9.4, который демонстрирует базовые принципы работы программы `sendmail`.

Итак, выполним последовательность команд и проанализируем полученный результат (лишние строки убраны):

```
# ps -ef|grep PPID;ps -ef|grep sendmail;sendmail -bp
UID  PID   PPID  C  STIME  TTY   TIME CMD
root 7665  7531  0  15:50  pts/1 00:00:00 grep PPID
root 3251  1     0  15:36  ?     00:00:00 sendmail: accepting connections
/var/spool/mqueue is empty
               Total requests: 0
```

Из листинга видно, что программа `sendmail` запущена как процесс-демон с идентификатором PID, равным 3251, и ожидает подключений клиентов. Команда

```
sendmail -bp
```

показывает, что очередь сообщений, формирующаяся в каталоге /var/spool/mqueue, в момент запуска пуста.

Посмотрим, как `sendmail` обработает два сообщения, отправленные пользователю root. Для этого выполним команды

```
# sendmail root < msg1
# sendmail root < msg2
```

Здесь msg1 и msg2 — текстовые файлы, содержимое которых используется для формирования текста сообщения. Запустив еще раз команду

```
# ps -ef|grep PPID;ps -ef|grep sendmail;sendmail -bp
```

получим такой результат (показаны только значащие строки):

```
UID  PID  PPID  C STIME  TTY   TIME CMD
root 3251 1     0 15:36  ?     00:00:00 sendmail: accepting connections
           /var/spool/mqueue (2 requests)
----Q-ID----- --Size-- ---Q-Time-----------Sender/Recipient-----------
k43G02nf010497  147    Wed May  3 16:00 <root@slax.example.net>
                                         <root@slax.example.net>
k43G01tQ010489  147    Wed May  3 16:00 <root@slax.example.net>
                                         <root@slax.example.net>
          Total requests: 2
```

Оба сообщения временно сохраняются в очереди сообщений с идентификаторами `k43G02nf010497` и `k43G01tQ010489` (поле `Q-ID`), имеют размер 147 байтов, а отправителем и получателем сообщения является пользователь root. Далее сообщения будут отправлены адресатам — это легко проверить, выполнив через несколько секунд командную строку

```
# ps -ef|grep PPID;ps -ef|grep sendmail;sendmail -bp
```

На экран дисплея выводится следующая информация:

```
UID  PID   PPID C STIME TTY   TIME CMD
root 10602 7531 0 16:00 pts/1 00:00:00 grep PPID
root 3251  1    0 15:36 ?     00:00:00 sendmail: accepting connections
root 10524 3251 0 16:00 ?     00:00:00 sendmail: ./k43G02nf010497 from queue
                /var/spool/mqueue (2 requests)
----Q-ID--------Size---------Q-Time-----------Sender/Recipient----
k43G02nf010497* 147    Wed May  3 16:00    <root@slax.example.net>
                                            <root@slax.example.net>
k43G01tQ010489  147    Wed May  3 16:00    <root@slax.example.net>
                                            <root@slax.example.net>
                Total requests: 2
```

На данном этапе началась отправка сообщения с идентификатором `k43G02nf010497`. Для этого процесс-демон `sendmail`, имеющий идентификатор PID, равный 3251, создал дочерний процесс с идентификатором PID, равным 10524, который собственно и выполняет отправку сообщения `k43G02nf010497`.

После обработки обоих сообщений каталог /var/spool/mqueue очищается, а дочерний процесс завершается.

Программу `sendmail` можно запустить, изменив второй параметр на `-qp`, как показано далее:

```
# sendmail -bd -qp
```

В этом случае для обработки очереди сообщений при запуске программы создается отдельный дочерний процесс, который выполняется постоянно и не уничтожается после окончания обработки текущих сообщений, находящихся в очереди. Он отслеживает поступающие в очередь сообщения с указанной периодичностью (если значение интервала времени не указано, как в данном примере, то по умолчанию оно принимается равным 1 секунде).

После запуска команды просмотр таблицы процессов дает следующий результат:

```
# ps -ef|grep sendmail
root   21564  1     0 16:35 ?  00:00:00 sendmail: accepting connections
root   21565 21564  0 16:35 ?  00:00:00 sendmail: running queue:
/var/spool/mqueue
```

Здесь дочерний процесс с идентификатором PID, равным 21565, создан основным процессом с идентификатором PID, равным 21654. Дочерний процесс выполняет обработку очереди сообщений и завершается только с завершением основного процесса.

Программа sendmail может использоваться для автоматизации отправки сообщений адресатам в командных файлах и приложениях на языках высокого уровня. Так, например, командный файл, исходный текст которого показан далее, проверяет наличие на диске текстового файла с именем newfile и, если таковой обнаружен, пересылает его содержимое пользователю root:

```
if [ -e newfile ]
  then
    cat newfile | sendmail root && rm newfile
fi
```

После успешной отправки сообщения файл newfile удаляется.

Более сложный пример использования программы sendmail написан на языке Perl. Приложение, исходный текст которого приводится далее, позволяет переслать содержимое открытого файла по электронной почте пользователю root:

```
#!/usr/bin/perl -w
$file = shift or die "Usage: $0 path\n";
open(FD, "$file") or die "open: $!";
@data = <FD>;
close(FD);
#
open(SENDMAIL, "|/usr/sbin/sendmail -oi root")
     or "sendmail: $!";
print SENDMAIL "@data.\n";
close(SENDMAIL);
```

Имя файла указывается в качестве единственного параметра программы. Анализ программы не вызывает трудностей — для этого достаточно знать основы программирования на языке Perl, изложенные в *главе 12*.

Следует отметить, что использование sendmail в приложениях, написанных на языках высокого уровня, значительно расширяет функциональные возможности таких программ по обмену данными с удаленными пользователями и другими, отличными от UNIX, системами.

Более подробную информацию о программе sendmail можно найти в man-страницах. Принципы конфигурирования sendmail достаточно хорошо описаны в документации, которую можно взять из Интернета.

9.4. Протоколы электронной почты

Для работы с электронной почтой через Интернет наиболее часто используются протоколы SMTP, POP3 и IMAP4. Мы рассмотрим особенности функционирования этих протоколов, а также их взаимосвязь, и начнем с SMTP (Simple Mail Transfer Protocol).

Протокол SMTP был разработан исключительно для взаимодействия почтовых серверов между собой. Применительно к модели взаимодействия открытых систем OSI SMTP находится на уровне приложений и взаимодействует только с TCP/IP, расположенным на транспортном уровне.

Протокол SMTP не поддерживает другие сетевые протоколы, но, тем не менее, это не стало препятствием на пути его широкого распространения. Более того, SMTP стал фактически стандартом для обмена электронной почтой в Интернете. В настоящее время все разработчики программного обеспечения для систем электронной почты поддерживают SMTP в качестве базового протокола или, по крайней мере, на уровне шлюзов. Столь широкая популярность этого протокола объясняется сравнительной простотой его реализации, широкими возможностями по масштабируемости, а также обратной совместимостью с существующими версиями почтовых систем.

Развитие систем электронной почты на базе SMTP происходит в следующих направлениях:

- расширение спецификаций для взаимодействия "сервер — сервер" (собственно SMTP);
- создание и развитие протоколов взаимодействия "клиент — сервер" (протоколы POP3, IMAP4);
- внедрение и развитие нового формата сообщений (MIME).

В первых версиях протокола SMTP поддерживался ограниченный набор команд и сервисов для приема и передачи сообщений. В настоящее время широко используется расширенный вариант (Extended SMTP, ESMTP), в который, кроме поддержки базовых функций, включена поддержка таких функций, как подтверждение доставки (Delivery Notification Request, DNR), синхронизация максимально допустимого размера сообщений, передаваемых между серверами, и принудительная инициация передачи накопленной почты. Тем не менее, одной из слабых сторон SMTP была и остается невозможность аутентификации входящих соединений и шифрования диалога и потока передачи данных между серверами.

Отсутствие средств аутентификации входящих соединений не позволяет использовать SMTP для обслуживания доступа клиентов. Стандартная система на базе SMTP требует прав доступа клиента к файлу своего почтового ящика для получения сообщений и манипуляций ими. Для реализации такого режима работы был разработан протокол обслуживания клиентской почты (Post Office Protocol, POP).

Наиболее распространена версия POP3, а последние реализации POP-протокола поддерживают аутентификацию с шифрованием имени и пароля,

а также шифрование трафика по протоколу SSL (Secure Socket Layer). К недостатку протокола POP3 можно отнести то, что нельзя просмотреть параметры сообщения непосредственно на сервере, а требуется загрузить его на клиентский компьютер. Для решения этой проблемы, а также других функциональных ограничений был разработан протокол IMAP4, который поддерживает большинство современных систем.

Следует заметить, что независимо от того, используется ли классический клиент (программа `mail`), или один из протоколов POP3 или IMAP4, отправка подготовленных клиентом сообщений требует функционирования сервера SMTP.

Изначально SMTP-системы были рассчитаны на передачу информации исключительно в текстовом виде (7-битовый набор символов), не позволяя передавать символы национальных алфавитов. Проблема передачи двоичных файлов была решена с появлением стандарта UUENCODE, который позволяет включать предварительно преобразованные из бинарной формы в текстовую произвольные данные непосредственно в текст сообщения. Тем не менее, данный протокол не решал до конца всех проблем, поскольку принимающая сторона не располагала информацией о вложении (типе данных и ассоциированном с ними приложении), что усложняло процесс обработки такого сообщения.

Таким образом, с развитием форматов данных назрела необходимость создания универсального формата типизации и представления двоичных данных и текста, содержащего национальные символы. Таким универсальным форматом стал набор многофункциональных расширений электронной почты Интернета (Multipurpose Internet Mail Extensions, MIME). Данный формат оказался на редкость удачным, поскольку в него были заложены возможности неограниченного расширения как поддерживаемых типов данных, так и кодировок национальных алфавитов.

Вернемся к протоколу SMTP и рассмотрим более подробно структуру сообщения. Применительно к сообщению можно сказать, что оно состоит из служебной (адресной) информации (конверта) и содержимого, состоящего, в свою очередь, из заголовка и тела. Состав полей в заголовке определяется форматом тела сообщения (UUENCODE или MIME), при этом ни одно из полей не является обязательным. Как правило, указываются поля "Кому" (`To:`), "От кого" (`From:`) и "Тема" (`Subject:`). Если используется формат MIME, то заголовок обязательно должен содержать строку "`MIME-Version: 1.0`". Полный перечень возможных полей в заголовке сообщения SMTP содержится в документе RFC 2076.

Одной из серьезных проблем при передаче сообщений посредством SMTP является низкий уровень конфиденциальности — сообщения передаются в текстовом виде и могут быть легко перехвачены и изменены. Проблему удалось решить, когда был разработан стандарт на шифрование тела сообщения под названием Secure MIME или S/MIME. Тем не менее, этот протокол не может защитить от перехвата заголовка сообщения.

Стандарты, на которых базируются SMTP-системы, опубликованы в виде рекомендаций (Reference for Comments, RFC) комитета по стандартам Интернета. Перейдем к практическим реализациям протоколов почтового обмена и начнем с SMTP.

9.4.1. Протокол SMTP

Протокол SMTP (Simple Mail Transfer Protocol, простой протокол передачи почты) поддерживает передачу сообщений между произвольными узлами Интернета. Обладая механизмами промежуточного хранения почты для повышения надежности доставки, протокол SMTP допускает использование различных транспортных служб и почтовых серверов. Он может работать даже в сетях, не поддерживающих стек протоколов TCP/IP. Протокол SMTP позволяет группировать сообщения, предназначенные одному адресату, и создавать копии электронных сообщений для передачи по разным адресам.

Передача сообщений по протоколу SMTP выполняется по схеме "клиент-сервер". При этом приложение, инициирующее запрос на отправку сообщения, является клиентом, а приложение, передающее сообщение, — сервером. Если сообщение передается между двумя серверами, то клиентом является очередной промежуточный почтовый сервер, получивший сообщение. Он же инициирует отправку сообщения дальше, пока не будет достигнут пункт назначения.

Программно протокол SMTP реализован в виде процесса-демона, принимающего входящие запросы на отправку сообщений на TCP-порту 25. Наиболее популярной реализацией протокола SMTP является программа `sendmail`, которую мы подробно рассматривали в предыдущем разделе. Существуют и другие программные реализации протокола SMTP, такие, например, как `postfix` и `qmail`. В некоторых операционных системах UNIX протокол SMTP реализован как отдельный процесс-демон под именем `smtpd`.

Перед настройкой или диагностированием сервера SMTP желательно убедиться, что запись об этой службе присутствует в файле /etc/services, например:

```
# cat /etc/services|grep smtp
smtp            25/tcp          mail
```

```
smtp            25/udp          mail
smtps           465/tcp         # SMTP over SSL (TLS)
```

Если по каким-либо причинам запись отсутствует, то нужно прописать соответствующую строку вручную.

Клиентская часть SMTP-протокола, так же как и у протокола POP3, может выполняться одной из почтовых программ. Спецификация протокола SMTP не предусматривает аутентификации, а это означает, что можно отправить сообщение от имени какого-либо пользователя, не зная его пароля.

Взаимодействие клиента и сервера (сеанс) начинается с приветствия, которое сервер отправляет клиенту при успешном установлении соединения. Сообщения, посылаемые клиентом серверу, называют командами, а сообщения, передаваемые сервером клиенту, — ответами. Среди команд клиента различают команды без параметров, команды с обязательными параметрами и команды с опциональными (необязательными) параметрами. Ответ сервера всегда начинается с трехзначного числа — кода ответа, после которого следует текстовое пояснение.

Умение работать с командами протокола SMTP очень полезно, особенно при инсталляции и настройке SMTP-сервера, а также для проверки работы клиентских программ. Нередко возникают и такие ситуации, когда проверку работоспособности почтовой системы и устранение неисправностей можно выполнить исключительно с помощью команд протоколов SMTP (это касается также протоколов POP3, IMAP4 и т. д.) непосредственно на сервере.

Далее приводится пример сеанса работы с сервером SMTP:

```
# telnet yuryhost 25
Trying ::1...
telnet: connect to address ::1: Network is unreachable
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 localhost ESMTP Sendmail 8.13.4+Sun/8.13.4; Wed, 15 Mar 2006 20:07:22 GMT
helo root
250 localhost Hello localhost [127.0.0.1], pleased to meet you
mail from:root
250 2.1.0 root... Sender ok
rcpt to:root
250 2.1.5 root... Recipient ok
data
354 Enter mail, end with "." on a line by itself
```

```
Test string to myself
.
250 2.0.0 k2FK7M3C000709 Message accepted for delivery
quit
221 2.0.0 localhost closing connection
Connection to localhost closed by foreign host.
```

Сеанс начинается с установки соединения с сервером — это можно сделать при помощи `telnet`:

```
# telnet yuryhost 25
Trying ::1...
telnet: connect to address ::1: Network is unreachable
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 localhost ESMTP Sendmail 8.13.4+Sun/8.13.4; Wed, 15 Mar 2006 20:07:22 GMT
```

Если вы видите подобные строки, то это означает, что клиент установил соединение с сервером SMTP. Обратите внимание, что в роли сервера SMTP выступает хорошо знакомая нам программа `sendmail`, о чем свидетельствует строка, начинающаяся с кода 220.

Далее выполняем команду `helo`. Она является необязательной, и клиент посылает ее серверу для того, чтобы убедиться, что взаимодействуют нужные хосты. В качестве параметра клиент должен указать свое доменное имя:

```
helo localhost
250 localhost Hello localhost [127.0.0.1], pleased to meet you
```

Далее начинаем создавать почтовое сообщение. Первое, что нужно сделать, — указать отправителя сообщения при помощи команды `mail from`:

```
mail from:root
250 2.1.0 root... Sender ok
```

В данном примере отправителем и получателем сообщения будет суперпользователь root, имя которого и указано в команде. Затем указываем получателя сообщения (команда `rcpt to`):

```
rcpt to:root
250 2.1.5 root... Recipient ok
```

Можно вводить несколько команд `rcpt to`, содержащих адреса разных получателей.

Далее следует ввести команду `data`, уведомляющую сервер, что будет набираться текст сообщения. После получения подтверждения можно набрать текст сообщения, закончив ввод символом точки:

```
data
354 Enter mail, end with "." on a line by itself
Test string to myself
.
```

После нажатия клавиши <Enter> сообщение отправляется адресату и выводится строка, подтверждающая отправку:

```
250 2.0.0 k2FK7M3C000709 Message accepted for delivery
```

Здесь `k2FK7M3C000709` — идентификатор сообщения. Сервер SMTP выполняет отправку одного почтового сообщения в одной транзакции, включающей выполнение команд `mail`, `rcpt` и `data`. Как только завершена передача тела сообщения, SMTP-транзакция считается завершенной. В течение одного сеанса можно отправить несколько сообщений, т. е. выполнить несколько транзакций.

Через некоторое время клиент получает уведомление о получении сообщения:

```
You have mail in /var/mail/root
```

Само сообщение можно прочитать командой `mail`:

```
# mail
From root@localhost Wed Mar 15 20:08:47 2006
Date: Wed, 15 Mar 2006 20:07:22 GMT
From: Super-User <root>
Message-Id: <200603152008.k2FK7M3C000709@localhost>
Content-Length: 22

Test string to myself

?
```

SMTP-протокол поддерживает еще одну команду — `help`, позволяющую получить справочную информацию об SMTP-сервере. Команда вводится без параметров и результат ее работы таков:

```
help
214-2.0.0 This is sendmail version 8.12.8
214-2.0.0 Topics:
214-2.0.0       HELO    EHLO    MAIL    RCPT    DATA
```

```
214-2.0.0       RSET    NOOP    QUIT    HELP    VRFY
214-2.0.0       EXPN    VERB    ETRN    DSN     AUTH
214-2.0.0       STARTTLS
214-2.0.0 For more info use "HELP <topic>".
214-2.0.0 To report bugs in the implementation send email to
214-2.0.0       sendmail-bugs@sendmail.org.
214-2.0.0 For local information send email to Postmaster at your site.
214 2.0.0 End of HELP info
```

Мы рассмотрели наиболее часто используемые команды протокола SMTP. Полное описание протокола SMTP приведено в спецификации RFC 821. Как видно из листинга сеанса связи, сервер SMTP при выполнении команд обязательно выдает код результата, который интерпретируется следующим образом:

- `220 <domain>` — сервер готов к взаимодействию (приветствие);
- `221 <domain>` — сервер закрывает канал передачи (перед отсоединением);
- `250` — положительный ответ на любую команду;
- `354` — сервер готов принимать сообщение (ответ на команду `data`);
- `421 <domain>` — сервис недоступен, канал передачи будет закрыт;
- `500` — синтаксическая ошибка (например, слишком длинная строка);
- `501` — синтаксическая ошибка в параметре;
- `502` — команда не поддерживается;
- `503` — неверная последовательность команд;
- `504` — параметр команды не поддерживается;
- `550` — отрицательный ответ на команду `rcpt` (например, указан несуществующий адресат);
- `554` — ошибка передачи.

Отправлять почтовые сообщения посредством командного интерфейса, предоставляемого SMTP-сервером, несколько неудобно, поэтому рассмотренные команды используют, в основном, для проверки работоспособности почтовых серверов. В рассмотренном ранее сеансе связи с сервером SMTP для установки соединения мы воспользовались протоколом telnet. Нередко протокол telnet отключают с целью повышения безопасности компьютера при работе в локальной сети или в Интернете. В этом случае воспользоваться программой telnet будет невозможно.

Если SMTP-сервер работает на локальном компьютере, то проверить его работоспособность можно и другими способами, например, просмотрев список выполняемых процессов или активных сетевых соединений, хотя и это не

дает полной уверенности. Если же сервер SMTP работает на удаленном компьютере, то проверить его работоспособность при отключенных службах удаленного доступа (в частности, telnet) может оказаться вообще невозможным. Тем не менее, для проверки работоспособности SMTP-сервера в подобных случаях можно использовать простую программу-клиента, работающую непосредственно с TCP-портом 25. Программа в качестве единственного аргумента принимает имя хоста, на котором работает сервер SMTP. Исходный текст данной программы (назовем ее `advtest_smtp`) представлен далее

```
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <netdb.h>

#define SMTPport 25

int SMTPcon, ret;
char buf[1024];
char *HELO = "HELO\n";
char *HELP = "HELP\n";
char *QUIT = "QUIT\n";

void get_answer()
{
  int ret = recv(SMTPcon, buf, sizeof(buf), 0);
  if (ret > 0)
  {
    buf[ret] = 0;
    printf("ANSWER: %s\n", buf);
  }
}

int main(int argc, char* argv[])
{
 if (argc != 2)
 {
   printf("Usage: %s hostname\n", argv[0]);
```

```
    exit(1);
  }
  struct sockaddr_in SMTPserver;
  struct hostent *host;
  bzero(&SMTPserver, sizeof(SMTPserver));
  host = gethostbyname(argv[1]);
  memcpy(&SMTPserver.sin_addr, host->h_addr_list[0], host->h_length);
  SMTPserver.sin_family = AF_INET;
  SMTPserver.sin_port = htons(SMTPport);
  SMTPcon = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
  connect(SMTPcon, (struct sockaddr*)&SMTPserver, sizeof(SMTPserver));
  get_answer();
  send(SMTPcon, HELO, strlen(HELO), 0);
  get_answer();
  send(SMTPcon, HELP, strlen(HELP), 0);
  get_answer();
  send(SMTPcon, QUIT, strlen(QUIT), 0);
  get_answer();
  close(SMTPcon);
  return 0;
}
```

В этой программе делается попытка подсоединиться к TCP-порту 25 сервера посредством операторов

```
SMTPcon = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
connect(SMTPcon, (struct sockaddr*)&SMTPserver, sizeof(SMTPserver));
```

После этого программа-клиент отправляет на сервер SMTP несколько команд:

```
send(SMTPcon, HELO, strlen(HELO), 0);
send(SMTPcon, HELP, strlen(HELP), 0);
send(SMTPcon, QUIT, strlen(QUIT), 0);
```

и получает ответы с помощью функции `get_answer()`.

По результатам ответов можно судить о работоспособности сервера. Результат работы программы может быть таким, как показано далее:

```
# ./advtest_smtp yuryhost
ANSWER: 220 localhost ESMTP Sendmail 8.13.4+Sun/8.13.4; Wed, 15 Mar 2006
11:57:28 GMT


ANSWER: 501 5.0.0 HELO requires domain address
```

```
ANSWER: 214-2.0.0 This is sendmail version 8.13.4+Sun
214-2.0.0 Topics:
214-2.0.0       HELO    EHLO    MAIL    RCPT    DATA
214-2.0.0       RSET    NOOP    QUIT    HELP    VRFY
214-2.0.0       EXPN    VERB    ETRN    DSN
214-2.0.0 For more info use "HELP <topic>".
214-2.0.0 To report bugs in the implementation contact Sun Microsystems
214-2.0.0 Technical Support.
214-2.0.0 For local information send email to Postmaster at your site.
214 2.0.0 End of HELP info

ANSWER: 221 2.0.0 localhost closing connection
```

9.4.2. Протокол POP3

В этом разделе мы рассмотрим другой, очень важный протокол, определяющий способы обработки входящих сообщений электронной почты, — протокол POP3 (Post Office Protocol).

В процессе функционирования POP3-сервер прослушивает TCP-порт 110, который используется по умолчанию. Применение протокола POP3 предполагает наличие клиента и сервера. Клиент создает TCP-соединение с сервером, после чего сервер отправляет приглашение и начинается сеанс работы.

В начале настройки или проверки сервера POP3 желательно убедиться, что запись об этой службе присутствует в файле /etc/services, например:

```
# cat /etc/services|grep pop3
pop3            110/tcp         pop-3           # POP version 3
pop3            110/udp         pop-3
pop3s           995/tcp                         # POP-3 over SSL
pop3s           995/udp                         # POP-3 over SSL
```

Если по каким-либо причинам служба не упоминается в данном файле, то можно прописать соответствующую строку вручную.

Клиент и POP3-сервер обмениваются информацией до тех пор, пока соединение не будет закрыто или прервано. Команды POP3, как и SMTP, состоят из ключевых слов, после которых могут следовать один или более аргументов. Все команды заканчиваются командами возврата каретки (CR) и перевода строки (LF).

Ключевые слова и аргументы состоят из печатаемых ASCII-символов, причем ключевое слово и аргументы разделены одиночным пробелом. Ключевое слово может состоять из 3-х или 4-х символов, а аргумент может быть дли-

ной до 40 символов. Ответы в POP3 состоят из индикатора состояния и ключевого слова, за которым может следовать дополнительная информация. Ответ заканчивается парой CR–LF.

Существуют только два индикатора состояния: +OK — положительный и -ERR — отрицательный. Ответы на некоторые команды могут состоять из нескольких строк. В этих случаях каждая строка разделена парой CR–LF, а в конец ответа помещается ASCII-символ с кодом 46 (.) и пара CR–LF.

POP3-сессия последовательно проходит несколько режимов. После установки соединения сервер отправляет приглашение клиенту, и сессия переходит в режим AUTHORIZATION (авторизация). В этом режиме клиент должен идентифицировать себя на сервере, указав свое имя (команда USER) и набрав пароль (команда PASS). После успешной идентификации сессия переходит в режим TRANSACTION (передача). В этом режиме клиент вводит команды, которые должны быть выполнены сервером.

После ввода команды QUIT сессия переходит в режим UPDATE — сервер POP3 освобождает все занятые ресурсы и завершает работу, закрывая TCP-соединение. POP3-сервер может автоматически закрывать соединение по истечении определенного интервала времени (прерывание по тайм-ауту), если соединение неактивно, т. е. не происходит никакого обмена данными. В этом случае срабатывает таймер, для которого значение тайм-аута определяется параметром INACTIVITY AUTOLOGOUT. Для большинства систем тайм-аут принимается равным 10 минутам.

Рассмотрим практический пример обмена информацией между клиентом и сервером POP3. Далее по тексту символом C обозначен клиент, а символом S — сервер POP3. Как только клиент установил соединение с POP3-сервером, тот отправляет приглашение, заканчивающееся комбинацией CR–LF:

```
S: +OK POP3 server ready
```

Затем POP3-сессия переходит в режим AUTHORIZATION. Клиент должен идентифицировать себя на сервере, используя команды USER и PASS.

Вначале следует ввести команду USER, после которой указать имя пользователя. В случае положительного ответа сервера следующей командой должна быть PASS, за которой нужно ввести пароль.

Если после отправки команды USER или PASS сервер выдает ошибку (-ERR), то можно выполнить еще одну попытку авторизации или выйти из сессии по команде QUIT. После успешной авторизации сервер открывает и блокирует почтовый ящик (maildrop), а также выводит на консоль информацию о количестве сообщений в почтовом ящике, а также об их размере. Теперь сессия находится в режиме TRANSACTION.

Рассмотрим команды протокола POP3 и начнем с команды USER. Ее синтаксис таков:

```
USER [имя]
```

В качестве параметра команда принимает имя почтового ящика. Результатом выполнения команды USER является один из ответов:

```
* +OK name is a valid mailbox
* -ERR never heard of mailbox name
```

Вот примеры удачного и неудачного выполнения команды USER:

```
C: USER yury
S: +OK yury is a real mailbox


C: USER user1
S: -ERR sorry, no mailbox for you here
```

После успешного выполнения команды USER следующей командой, которую следует выполнить, является команда PASS:

```
PASS [пароль]
```

Параметром команды является пароль почтового ящика, который передается серверу. Возможны следующие варианты ответов:

```
* +OK maildrop locked and ready
* -ERR invalid password
* -ERR unable to lock maildrop
```

Вот результаты выполнения команды PASS:

```
C: USER yury
S: +OK yury is a real mailbox
C: PASS mymail
S: +OK yury's maildrop has 2 messages (320 octets)


C: USER yury
S: +OK yury is a real mailbox
C: PASS mymail
S: -ERR maildrop already locked
```

Команда QUIT завершает работу POP3-сессии, при этом сервер переходит в режим UPDATE. Возможный вариант ответа:

```
* +OK
```

Пример выполнения команды `QUIT` приведен далее:

```
C: QUIT
S: +OK dewey POP3 server signing off
```

Перейдем к анализу группы основных команд, выполняющих обработку электронной почты пользователя. После успешной идентификации пользователя на сервере POP3-сессия переходит в режим `TRANSACTION`, где пользователь может выполнять очередные команды, получая от сервера ответы. Далее приводятся примеры некоторых доступных команд в этом режиме и возможные ответы сервера.

Например, получив команду `STAT`, сервер выдает положительный ответ `+OK`, за которым следует количество сообщений в почтовом ящике и их общий размер в байтах. Сообщения, отмеченные для удаления, в ответе сервера не учитываются. Возможный ответ:

```
* +OK n s
```

Пример использования команды `STAT`:

```
C: STAT
S: +OK 2 320
```

Команда

```
LIST [сообщение]
```

в качестве необязательного аргумента может принимать номер сообщения. Если аргумент присутствует, то сервер выдает информацию об указанном сообщении. Если аргумент не был передан, то сервер выводит информацию обо всех сообщениях, находящихся в почтовом ящике. Сообщения, отмеченные для удаления, в списке не перечисляются. Возможным ответом может быть один из следующих:

```
* +OK scan listing follows
* -ERR no such message
```

Далее приводится несколько примеров:

```
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: LIST 2
S: +OK 2 200

C: LIST 3
S: -ERR no such message, only 2 messages in maildrop
```

По команде

```
RETR [сообщение]
```

где параметр является номером сообщения, сервер передает содержание сообщения. Возможные ответы:

```
* +OK message follows
* -ERR no such message
```

Пример выполнения команды RETR:

```
C: RETR 1
S: +OK 120 octets
S:
S: .
```

Следующая команда, которую мы проанализируем, — это команда DELE. Она имеет синтаксис:

```
DELE [сообщение]
```

В качестве аргумента команда принимает номер сообщения. POP3-сервер отмечает указанное сообщение для удаления, но не удаляет его, пока сессия не перейдет в режим UPDATE. Возможные ответы:

```
* +OK message deleted
* -ERR no such message
```

Примеры вызова команды DELE и результаты выполнения:

```
C: DELE 1
S: +OK message 1 deleted


C: DELE 2
S: -ERR message 2 already deleted
```

Команда NOOP не имеет аргументов, в ответ на нее сервер не выполняет никаких действий и отвечает всегда положительно:

```
* +OK
```

Пример:

```
C: NOOP
S: +OK
```

Следующая команда, которую мы рассмотрим, — команда RSET. Она не имеет аргументов и снимает метку с сообщений, которые были отмечены для удаления. Возможный ответ:

```
* +OK
```

Пример применения:

```
C: RSET
S: +OK maildrop has 2 messages (320 octets)
```

Когда клиент передает команду `QUIT` в режиме `TRANSACTION`, сессия переходит в режим `UPDATE`. В этом режиме сервер удаляет все сообщения, помеченные для удаления. После этого TCP-соединение закрывается.

Следующая группа команд выполняет обработку составных частей сообщения и позволяет компоновать сообщения по усмотрению пользователя. Рассмотрим вначале команду `TOP`, которая имеет следующий синтаксис:

```
TOP [сообщение] [n]
```

Аргумент `сообщение` указывает на номер сообщения, аргумент `n` — положительное число. По этой команде POP3-сервер передает заголовок сообщения и указанное количество строк из тела сообщения (если ответ положительный) или сообщает об ошибке (если, например, сообщение не обнаружено). Возможные ответы:

```
+OK top of message follows
-ERR no such message
```

Следующие примеры демонстрируют применение команды `TOP`:

```
C: TOP 1 10 S: +OK
S: <здесь POP3-сервер передает заголовок
    первого сообщения и первые 10 строк из тела сообщения.>
S: .
C: TOP 100 3
S: -ERR no such message
```

Команда

```
UIDL [сообщение]
```

требует от сервера выдачи уникального идентификатора сообщения. Параметр представляет собой номер сообщения. Команда может выполняться без аргументов. В этом случае идентификаторы перечисляются для всех сообщений, кроме тех, которые отмечены для удаления. Возможные ответы:

```
+OK unique-id listing follows
-ERR no such message
```

Примеры использования команды `UIDL`:

```
C: UIDL S: +OK
S: 1 whqtswO00WBw418f9t5JxYwZ
```

```
S: 2 QhdPYR:00WBw1Ph7x7
S: . ...
C: UIDL 2
S: +OK 2 QhdPYR:00WBw1Ph7x7 ...
C: UIDL 3
S: -ERR no such message, only 2 messages in maildrop
```

Обобщая рассмотренный материал по системе команд POP3-сервера, приведу пример простого сеанса с POP3-сервером:

```
S: <создаем новое TCP-соединение с POP3-сервером через порт 110>
S: +OK POP3 server ready
C: USER yury
S: +OK User yury is exists
C: PASS mymail
S: +OK yury's maildrop has 2 messages (320 octets)
C: STAT
S: +OK 2 320
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: RETR 1
S: +OK 120 octets
S:
S: .
C: DELE 1
S: +OK message 1 deleted
C: RETR 2
S: +OK 200 octets
S:
S: .
C: DELE 2
S: +OK message 2 deleted
C: QUIT
S: +OK dewey POP3 server signing off (maildrop empty)
C: <закрываем соединение>
```

Более подробная информация по протоколу POP3 находится на многочисленных интернет-сайтах.

9.4.3. Протокол IMAP4

Протокол обработки полученных сообщений IMAP4 по своим возможностям очень похож на POP3, но более надежен, к тому же обладает более широкими возможностями по управлению процессом обмена с сервером.

Протокол IMAP4 (Internet Message Access Protocol), как и POP3, позволяет клиентам получать доступ и манипулировать сообщениями электронной почты на сервере. Существенным отличием протокола IMAP4 от POP3 является то, что IMAP4 поддерживает работу с системой каталогов (или папок) на удаленных хостах так, как если бы они располагались на локальном компьютере. IMAP4 позволяет клиенту создавать, удалять и переименовывать почтовые ящики, проверять наличие новых сообщений и удалять старые.

Поскольку IMAP4 поддерживает механизм уникальной идентификации каждого сообщения в почтовой папке клиента, то он позволяет читать из почтового ящика только сообщения, удовлетворяющие определенным условиям, менять атрибуты сообщений и перемещать отдельные сообщения. Структура папок в значительной степени зависит от типа почтовой системы, но в любой системе у клиента есть специальный каталог INBOX, куда попадают поступающие ему сообщения.

Протокол IMAP4 работает поверх транспортного протокола TCP, который обеспечивает надежный канал передачи данных между клиентом и сервером IMAP4. При работе по TCP протокол IMAP4 использует порт 143. Команды и данные IMAP4 передаются по транспортному протоколу в том виде, в каком их отправляет сервер или пользователь. Перед настройкой или проверкой сервера IMAP4 желательно убедиться, что запись об этой службе присутствует в файле /etc/services, например:

```
# cat /etc/services|grep imap*
imap            143/tcp         imap2           # Interim Mail Access
Proto v2
imap            143/udp         imap2
imap3           220/tcp                         # Interactive Mail Access
imap3           220/udp                         # Protocol v3
imaps           993/tcp                         # IMAP over SSL
imaps           993/udp                         # IMAP over SSL
```

Если по каким-либо причинам служба не упоминается в данном файле, то можно прописать соответствующую строку вручную.

Принцип передачи данных IMAP4 такой же, как и у других подобных протоколов. Сначала клиент и сервер обмениваются приветствиями, затем клиент отправляет на сервер команды и данные. Сервер, соответственно, передает

клиенту ответы, являющиеся результатами обработки команд и данных. После завершения обмена соединение закрывается.

Если сервер использует таймер контроля времени соединения, то его нужно установить не менее чем на 30-минутный интервал.

Каждая команда клиента начинается с идентификатора или тега команды. Тег обычно представляет собой короткую строку, состоящую из букв и цифр, например, A0001, A0002 и т. д. Он является уникальным идентификатором данной команды клиента, причем ответы сервера или же следующие команды клиента могут ссылаться на команду по ее тегу.

Каждая команда клиента начинается с новой строки, кроме случаев, когда она передает поток данных заданной длины или требует ответа с сервера для продолжения работы (например, при аутентификации). В таких случаях команда может занимать несколько строк.

Строки данных, передаваемые сервером клиенту, могут вместо тега содержать символ `*`. Это означает, что они являются промежуточными строками в потоке данных, а идентификатор их команды находится в последней строке потока.

Если сервер обнаружил ошибку в команде, он отправляет уведомление `BAD` клиенту с тегом неправильной команды. Если команда успешно обработана — возвращается уведомление `OK` с тегом команды. Если команда вернула отрицательный результат, например, при невозможности ее выполнения — возвращается уведомление `NO` с тегом невыполненной команды.

Важной особенностью протокола IMAP4 является то, что взаимодействие клиента с сервером осуществляется по асинхронному принципу: клиент может отправить новую команду на сервер, ожидая ответ на предыдущую, если эти команды не взаимосвязаны, или ответ одной не повлияет на результат другой.

Сервер может обрабатывать несколько команд одновременно и отвечать на каждую из них по окончании ее обработки. При этом ответ на команду, поступившую позже, может поступить раньше. Ответ сервера всегда содержит тег той команды, к которой он относится.

При работе в асинхронном режиме клиент и сервер должны фиксировать весь поток данных, поскольку и сервер, и клиент в своих запросах и ответах могут ссылаться на команды и данные, введенные на предыдущих стадиях сессии обмена. Для того чтобы обеспечить гибкость и многофункциональность в обработке сообщений, почтовые системы IMAP4 присваивают сообщениям определенные атрибуты.

Каждое сообщение в почтовой системе, включающей IMAP4, имеет уникальный 32-разрядный идентификатор (UID), по которому можно получить к нему доступ. Любому сообщению, находящемуся в папке, присваивается следующее большее значение из уже занятых UID. Уникальные идентификаторы сообщений сохраняются от сессии к сессии и могут использоваться, например, для синхронизации каталогов мобильных пользователей.

Кроме уникального идентификатора, сообщение в системе IMAP4 имеет свой порядковый номер, т. е. все сообщения в данном почтовом ящике последовательно нумеруются. Если в почтовый ящик добавляется новое сообщение, ему присваивается номер на единицу больший, чем количество сообщений, находящихся в этом ящике. При удалении какого-либо сообщения из данной папки порядковые номера всех сообщений пересчитываются, поэтому порядковый номер сообщения может меняться во время сессии. Большинство команд IMAP4 работает с порядковыми номерами сообщений, а не с UID.

Помимо числовых идентификаторов, сообщениям назначаются флаги. Одни флаги могут быть действительны для данного сообщения постоянно от сессии к сессии, другие — только в данной сессии. Наиболее часто применяются следующие флаги:

- `\Seen` — данное сообщение было прочитано;
- `\Answered` — на сообщение был дан ответ;
- `\Deleted` — сообщение отмечено для удаления;
- `\Draft` — формирование данного сообщения еще не завершено;
- `\Recent` — сообщение только что поступило в почтовый ящик, т. е. данная сессия — первая, в которой можно прочитать это сообщение.

Кроме того, на сервере IMAP4 хранятся дата и время получения сообщения сервером. Например, если сообщение получено по SMTP, то фиксируются дата и время доставки по адресу назначения, общий размер сообщения и структура сообщения в соответствии со спецификацией MIME.

Как уже было отмечено, IMAP4 — гибкий и многофункциональный протокол с широкими возможностями. Он обслуживает более 20 различных команд клиента по управлению состоянием своей почты. Далее будут описаны только некоторые из клиентских команд, и на примерах их обработки будет показана общая схема взаимодействия клиента и сервера IMAP4.

После установления соединения с сервером и получения строки приветствия `OK` клиент должен зарегистрироваться в системе, для чего и используется команда `LOGIN`. В качестве аргумента она принимает строку с идентификатором и паролем клиента. Следующий пример демонстрирует использование

данной команды: здесь символом C обозначен клиент, а символом S — сервер IMAP4:

```
S:   * OK IMAP4 revl Service Ready
C:   aOOl login pop q1w2e3
S;   aOOl OK LOGIN completed
```

Команда LOGIN передает пароль и идентификатор пользователя по сети в открытом виде. Если пользователю необходима защита информации, находящейся в его почте, нужно использовать команду AUTHENTICATE. Аргументом команды является строка, указывающая механизм аутентификации, который применяет пользователь. Дальнейший обмен между клиентом и сервером будет зависеть от выбранного типа аутентификации. Вот как может выглядеть процесс аутентификации при использовании механизма шифрования KERBEROS:

```
S: * OK KerberosV4 IMAP4revl Server
C: AO 01 AUTHENTICATE KERBEROS_V4
S:  + AmFYig==
C: BAcAQrJ5EUkVXLkNNVS5FRFUAOCAsho84kLN3/IJmrMG+25a4DT
+nZIiriJjnTNHJUtxAA+oOKPKfHEcAFs9a3CL50ebe/ydHJUwYFd
WwuQlMWiy6IesKvjL5rL9WjXUb9MwT9bpObYLGOKilQh
S: + or//EoAADZI=
C: DiAF5MgA+oOIALuBkAAmw==
S: A001 OK Kerberos V4 authentication successful
```

После регистрации в системе клиент должен выбрать каталог сообщений, с которым он будет работать. Выбор каталога осуществляется с помощью команды SELECT, аргументом которой является имя почтового каталога. Команда SELECT может работать так, как показано в примере:

```
C A142 SELECT INBOX
S * 172 EXISTS
S * 1 RECENT
S * OK [UNSEEN 12) Message 12 is first unseen
S * OK [UIDVALIDITY 3857529045] UIDs valid
S * FLAGS (\Answered \Flagged \Deleted \Seen \Draft)
S * OK [PERMANENTFLAGS (\Deleted \Seen \*)] Limited
S A142 OK [READ-WRITE] SELECT completed
```

Сервер IMAP4, прежде чем подтвердить завершение обработки команды, передает клиенту атрибуты данного каталога. Еще раз посмотрим на приведенный пример. В папке INBOX находится 172 сообщения (строка "* 172 EXISTS"), одно из них только что поступившее (строка "* 1 RECENT").

В папке есть непрочитанные сообщения, при этом минимальный порядковый номер непрочитанного сообщения — 12 (строка "`* OK [UNSEEN 12] Message 12 is first unseen`"). Уникальный временный идентификатор папки INBOX в данной сессии — `3857529045` (строка "`* OK [UIDVAL1DITY 3857529045] UIDs valid`"). Сообщения в данной папке могут иметь флаги, указанные в строке `FLAGS` ("`* FLAGS (\Answered \Flageed \Deleted \Seen \Draft")`).

Клиент может менять у сообщений флаги `\Deleted` и `\Seen` (строка "`* OK [PERMANENTFLAGS (\Deleted \Seen \*)] Limited`").

Клиент имеет права на запись и чтение сообщений из INBOX (строка "`A142 OK [READ-WRITE] SELECT completed`").

Команда `SELECT` устанавливает текущий каталог для работы клиента. Если пользователю необходимо получить информацию о состоянии какого-либо каталога, достаточно применить команду `EXAMINE` с именем каталога в качестве аргумента команды:

```
C: A932 EXAMINE bloop
S: * 17 EXISTS
```

Команда `EXAMINE` возвращает те же значения, что и `SELECT`, но отличается от последней тем, что открывает указанный почтовый ящик только для чтения.

Если необходимо запросить статус какой-либо папки, не меняя текущий каталог, можно воспользоваться командой `STATUS`. В качестве параметров команда принимает имя папки и тип запрашиваемой информации. В зависимости от указанного типа команда может возвращать: количество сообщений в папке, количество новых сообщений, количество непрочитанных сообщений, UID следующего сообщения, которое будет добавлено в данную папку, например:

```
<>ttC: Д042 STATUS blob (MESSAGES UNSEEN)
S: * STATUS blob (MESSAGES 231 UNSEEN 12)
S: A042 OK STATUS completed
```

Для получения списка папок (подкаталогов), находящихся в определенной папке и доступных клиенту, можно воспользоваться командой `LIST`. В качестве аргументов команда принимает имя каталога, список подкаталогов, который нужно получить (пустая строка означает текущий каталог), и маску имен подкаталогов. При этом имена каталогов и маски имен подкаталогов могут интерпретироваться различным образом, в зависимости от реализации почтовой системы и иерархии папок. Например, список папок, находящихся в корневом каталоге, можно получить с помощью такой команды:

```
C: A004 LIST "/" *
S: * LIST (\Noinferiors ) "/" INBOX
```

```
S: * LIST <\Noinferiors ) "/" OUTBOX
S: * LIST <\Noinferiors ) "/".. WasteBox
S: A004 OK LIST completed
```

В ответе сервера содержится список папок в соответствии с их положением в иерархии и флаги данных папок (флаг `\Noinferiors` означает, что внутри данной папки нет иерархии и она невозможна).

После получения информации о каталоге пользователь может прочитать любое сообщение или определенную группу сообщений, а также часть сообщения или атрибуты конкретного сообщения. Для этого можно воспользоваться командой `FETCH`. В качестве аргументов команда принимает порядковый номер сообщения и критерии запроса, содержащие описание вида возвращаемой информации. Можно запросить часть заголовка или идентификатор сообщения в папке. Так, например, запрос заголовков сообщений, находящихся в папке INBOX с порядковыми номерами от 10 до 12, мог бы выглядеть так:

```
C: A654 FETCH 10:12 BODY [HEADER]
S: * 10 FETCH (BODY [HEADER] {350}
S: Date: Wed, 17 Jul 1996 02:23:25 -0700 (PDTl
S: From: raan@globe.com
S: Subject: Hi
S: To: imap@world.edu
S: Message-Id:
S^ mime-Vresion: 1.0
S: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
S:
S: )
S: *11 FETCH ....
S: *12 FETCH ....
S: A654 OK FETCH completed
```

Просмотрев сообщение, пользователь может сохранить его, добавить или удалить флаги, например, отметить данное сообщение для удаления — эти манипуляции можно выполнить с помощью команды `STORE`. В качестве аргументов команда принимает номера сообщений, идентификатор операции и перечень флагов. Например, операция добавления флага удаления (`\Deleted`) трем сообщениям выглядит следующим образом:

```
C: AOO3 STORE 2:4 +FLAGS (\DELETED)
S: *2 FETCH FLAGS (\Deleted \ Seen)
S: *3 FETCH FLAGS (\Deleted )
S: *4 FETCH FLAGS (\Deleted \Flagged \Seen)
S: A003 OK STORE completed
```

В ответ на выполнение команды будут переданы строки новых флагов указанных сообщений.

Клиент может выполнить поиск сообщений по определенным критериям при помощи команды `SEARCH`. Критерий поиска представляет собой комбинацию нескольких условий поиска, а результатом поиска будет группа сообщений, удовлетворяющих общим условиям. Можно указывать условия, относящиеся к составу, структуре тела или заголовку сообщений, к флагам, идентификаторам, датам сообщений. Результатом работы команды является строка из последовательных номеров сообщений, удовлетворяющих критерию поиска. Например, поиск всех непрочитанных сообщений, поступивших от `smith` с 1 марта 2002 года, может выглядеть так:

```
C: A282 SEARCH UNSEEN FROM 'Smith" SINCE 1-Mar-2002
S: * SEARCH 29 174 982
S: A282 OK SEARCH completed
```

Как видно, критерию поиска удовлетворяют сообщения с последовательными номерами 29, 174 и 982.

Протокол IMAP4 позволяет не только искать и читать сообщения в каталогах, но также добавлять, копировать и перемещать сообщения в каталоги (команда `APPEND`):

```
C: A003 APPENDSAVED-MESSAGES (\Seen) {310}
C: Date: Mon, 7 Feb 1997 21:52:25 - 0800 {PST}
C: From: Fred Foobar
C: Subject: aftenoon meeteng
C: TO: mooch@owatagu.siam.edu
C: Message-Id:
C: Mime-Version: 1.0
C: Content-Type: Text/PLAIN; CHARSET=US-ASCII
C:
C: Hello Joe, do you think we can meet at 3:30 tomorrow?
C:
S: A003 OK APPEND completed
```

Команда `COPY` копирует сообщения с заданными порядковыми номерами в указанный каталог, например:

```
C: A003 COPY 2:4 MEETENG
S: A003 OK COPY completed
```

Далее показан пример взаимодействия по протоколу IMAP4:

```
OK IMAP2 Server Ready
```

```
A001 LOGIN Fred Secret
A001 OK User Fred logged in
A002 SELECT INBOX
* FLAGS (Meeting Notice \Answered \Flagged \Deleted \Seen)
* 19 Exists
* 2 Recent
* A002 OK Select compete
A003 FETCH 1:19 ALL
* 1 Fetch ( .....
* 19 Fetch (....
A003 OK Fetch complete
A004 LOGOUT
* Bye IMAP2 server quitting
A004 OK Logout complete
```

Хочу заметить, что рассмотренные протоколы обмена с помощью электронной почты (SMTP, POP3, IMAP4) могут быть реализованы в той или иной степени во всех операционных системах UNIX. Это означает, что для работы электронной почты можно использовать как минимальный набор базовых программных средств, входящих в состав любой операционной системы, так и программное обеспечение, осуществляющее полное управление электронной почтой.

В заключение я приведу исходный текст программы, позволяющей протестировать работоспособность протоколов SMTP, POP3 и IMAP4 без протокола telnet. Программа (она называется `check_proto`) принимает в качестве единственного аргумента номер TCP-порта, с которым работает данный протокол:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <fcntl.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char* argv[])
{
```

```
 if (argc != 3)
 {
   printf("Usage: %s host port\n", argv[0]);
   exit(1);
 }
 int fdsock, ret, cnt;
 struct sockaddr_in client;
 int port = atoi(argv[2]);
 char buf[1024];
 bzero(&client, sizeof(client));
 inet_aton(argv[1], &client.sin_addr);
 client.sin_family = AF_INET;
 client.sin_port = htons(port);
 fdsock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
 connect(fdsock, (struct sockaddr*)&client, sizeof(client));
 ret = 0;
 for(cnt = 0; (cnt < 4) && (ret == 0); cnt++)
 {
   ret = recv(fdsock, buf, sizeof(buf), 0);
   if (ret > 0)
   {
     buf[ret] = 0;
     printf("ANSWER: %s\n", buf);
     cnt = 0;
     break;
   }
   sleep(1);
 };
 if (cnt > 3) printf("no answer from Server\n");
 close(fdsock);
 return 0;
}
```

Программа устанавливает соединение с соответствующим сервером (SMTP, POP3, IMAP4) посредством выполнения операторов

```
fdsock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
connect(fdsock, (struct sockaddr*)&client, sizeof(client));
```

после чего выводит полученную от сервера строку приглашения на экран. Вот пример работы программы `check_proto`:

```
# ./check_proto yuryhost 25
```

```
ANSWER: 220 localhost.localdomain ESMTP Sendmail 8.12.8/8.12.8; Sun, 12
Mar 2006 14:29:24 +0200
# ./check_proto yuryhost 110
ANSWER: +OK POP3 yuryhost v2001.78rh server ready
# ./check_proto yuryhost 143
ANSWER: * OK [CAPABILITY IMAP4REV1 LOGIN-REFERRALS STARTTLS AUTH=LOGIN]
yuryhost IMAP4rev1 2001.315rh at Sun, 12 Mar 2006 14:29:31 +0200 (EET)
```

9.4.4. MIME

На данный момент большинство систем электронной почты поддерживает спецификацию MIME (Multipurpose Internet Mail Extension), позволяющую расширить возможности передачи данных различного типа. Рассмотрим особенности этой спецификации.

Протокол MIME не является альтернативой протоколам TCP/IP для электронной почты, а только дополняет их. Данный протокол описывает спецификации содержимого сообщений, а не способы их доставки, как в протоколах SMTP, POP3, IMAP4 и др.

Стандарт MIME (документ RFC 1341) предназначен для описания тела почтового сообщения. До появления MIME использовался стандарт почтового сообщения ARPA (Standard for the Format of ARPA Internet Text Messages), описанный в документе RFC 822. ARPA был разработан для обмена текстовыми сообщениями относительно давно и к настоящему времени уже устарел, поскольку не соответствует современному уровню развития аппаратных средств и телекоммуникаций.

Согласно этому протоколу, в тело сообщения нельзя включать графику, аудио, видео и другие типы информации. Стандарт RFC 822 не предоставляет возможностей для передачи даже текстовой информации, поскольку это нельзя реализовать в 7-битовой кодировке ASCII. Ограничения RFC 822 становятся еще более очевидными, когда речь заходит об обмене сообщениями между разными почтовыми системами.

Стандарт MIME отличается от стандарта RFC 822. Если последний подробно описывает в заголовке почтового сообщения текстовое тело письма и механизм его рассылки, то MIME предназначен для описания в заголовке письма структуры тела почтового сообщения и обеспечивает возможности составления письма из информационных единиц различных типов.

Стандарт предусматривает несколько вариантов представления разнородной информации. Для этого используются специальные поля в заголовке почтового сообщения:

- поле версии MIME идентифицирует сообщение, подготовленное в новом стандарте;

- поле описания типа информации в теле сообщения позволяет обеспечить правильную интерпретацию данных;
- поле типа кодировки информации в теле сообщения указывает на тип процедуры декодирования;
- два дополнительных поля зарезервированы для более детального описания тела сообщения.

Стандарт MIME разработан как расширяемая спецификация, в которой подразумевается, что количество типов данных будет увеличиваться по мере развития форм представления данных, но неконтролируемый рост числа типов тоже недопустим. Каждый новый тип в обязательном порядке должен быть зарегистрирован в IANA (Internet Assigned Numbers Authority).

Рассмотрим более подробно форму и назначение полей, определяемых стандартом MIME.

Поле версии MIME указывается в заголовке почтового сообщения и позволяет определить программе рассылки почты, что сообщение подготовлено с использованием стандарта MIME. Формат поля таков:

```
MIME-Version: 1.0
```

Поле версии указывается в общем заголовке почтового сообщения и относится ко всему сообщению целиком. Следует заметить, что, в отличие от стандарта RFC 822, в стандарте MIME допускается смешивание поля заголовка сообщения с телом сообщения. Поэтому все поля разделены на два класса: общие поля заголовка, которые включаются в начало почтового сообщения, и частные поля заголовка, относящиеся только к отдельным частям составного сообщения и расположенные перед ними.

Поле типа содержания тела почтового сообщения (`Content-Type`) описывает тип данных, содержащихся в теле почтового сообщения. Это поле указывает программе чтения почты, какого рода преобразования необходимы для правильной интерпретации сообщения. Эту же информацию использует и программа рассылки при выполнении процедуры кодирования/декодирования почты. Стандарт MIME определяет семь типов данных, которые можно включать в тело письма:

- текст (`text`);
- смешанный тип (`multipart`);
- почтовое сообщение (`message`);
- графический образ (`image`);

- аудиоинформация (`audio`);
- фильм или видео (`video`);
- приложение (`application`).

Среди этих типов данных можно выделить четыре, предназначенные для описания нетекстовой информации:

- `image` — для описания графических образов. Наиболее часто используются файлы форматов GIF и JPEG;
- `audio` — для описания аудиоинформации;
- `video` — для передачи видеоинформации (AVI-файлов, MPEG и др.);
- `application` — для передачи данных любого другого формата, обычно служит для передачи двоичных данных с целью последующего промежуточного преобразования.

Назначение перечисленных типов следует из их названия. Многие данные передаются по почте в их исходном виде. Это могут быть 7-битовые или 8-битовые символы, 64base-символы и т. п. Однако при работе в разнородных почтовых средах необходимо определить механизм их представления в стандартном виде US ASCII. Для этого существуют процедуры кодирования такого рода данных. Наиболее широко применяется программа `uuencode`. Для того чтобы при получении данные были правильно распакованы, в стандарт введено поле `Content-Transfer-Encoding`. Синтаксис этого поля следующий:

```
Content-Transfer-Encoding:= "BASE64" | "QUOTED-PRINTABLE" |
                            "8BIT"   | "7BIT" |
                            "BINARY" | x-token
```

Каждая из альтернатив может применяться в подходящем случае. Альтернативы `8BIT`, `7BIT`, `BINARY` реально никакого преобразования не требуют, т. к. почта передается как поток байтов, и SMTP не различает такие данные. Однако они введены для строгости описания типов. `BASE64` обычно используется вместе с типом `text/ISO-8859-1`. `x-token` разрешает пользователю определить свою процедуру преобразования.

Как уже упоминалось ранее, стандарт определяет еще два дополнительных поля: `Content-ID` и `Content-Description`. Первое поле задает уникальный идентификатор содержания, а второе предназначено для комментария содержания. Программы просмотра эти поля обычно не отображают.

9.5. Программы для работы с электронной почтой

К настоящему времени разработано много программ, предназначенных для обслуживания систем электронной почты. Некоторые из них обладают расширенными возможностями по сравнению с классическими утилитами (`mail`, `sendmail`) и улучшенным интерфейсом пользователя. Среди всего многообразия таких программ рассмотрим наиболее популярные — `Pine`, `Postfix` и `fetchmail`.

Программа `Pine` является одной из самых распространенных почтовых программ для UNIX. Это экранная программа, и с ее помощью можно выполнять дополнительную обработку почтовых сообщений, что недоступно для простейших программ наподобие `mail`. Программа `Pine` позволяет выполнять такие манипуляции почтовыми сообщениями:

- обработка групп писем — можно отметить несколько писем в списке и отправить их все вместе нужному адресату;
- получение не только локальной почты, но и с серверов IMAP4 и POP3.

Кроме почты, `Pine` умеет также работать с группами новостей (Usenet news). В программе `Pine` можно пользоваться адресной книгой — это позволяет упростить работу, особенно при большом количестве адресатов. В `Pine` можно создавать электронную подпись (сигнатуру) — короткий текст, который будет автоматически добавляться в конец каждого письма.

Программа `Pine` обладает и целым рядом других возможностей, подробное описание которых можно найти в Интернете.

Программа `Postfix` является очень мощной почтовой системой, пользующейся весьма большой популярностью. В основе построения `Postfix` лежит модульный принцип. Сама программа представляет собой набор независимых резидентных модулей. Исполняемые модули постоянно присутствуют в памяти, и каждый из них может независимо использовать ресурсы других модулей. Рассмотрим коротко, как можно настроить `Postfix` для доставки писем в предположении, что программа уже проинсталлирована. Все настройки выполним для пользователя yury.

Прежде всего, следует убедиться в том, что для отправки почтовых сообщений используется протокол SMTP и что указан почтовый сервер, через который отправляется почта. Все эти настройки можно обнаружить в файле /etc/postfix/mail.cf. При необходимости нужно изменить или добавить параметры, если они не указаны:

```
defer_transport=smtp
relayhost = mail.ukrtel.net
```

Для отправки почты в данном случае используется сервер **mail.ck.ukrtel.net**. Для получения писем обратимся к программе `fetchmail`, которая будет рассмотрена далее. В файл .fetchmailrc необходимо добавить настройки для пользователя yury:

```
set postmaster "postmaster"
set bouncemail
set no spambounce
poll mail.ukrtel.net with proto POP3
       user 'yury' there with password 'passwdyury' is yury here
```

После этого следует перезапустить `Postfix`. Настройка для пользователя yury завершена. Для более детального ознакомления с возможностями программы `Postfix` можно обратиться за помощью к online-документации.

Одной из широко распространенных программ является `fetchmail`. Программа `fetchmail` — это утилита приема и пересылки почты, выполняющая загрузку почты с удаленных почтовых серверов с последующей ее передачей локальной системе доставки почты на машине клиента. После этого можно читать почту, используя обычные почтовые агенты, например, `mutt`, `elm` или `mail`. По сравнению с другими программами, в том числе и с `Pine`, `fetchmail` обладает более развитыми средствами для обработки почтовых сообщений.

Программа `fetchmail` сложнее в настройке, чем `Pine`, но и возможностей у нее больше. Утилита `fetchmail` может быть запущена в режиме демона для регулярной проверки и приема почты с одного или нескольких серверов. Программа `fetchmail` имеет следующий синтаксис:

`fetchmail [`*опция*`...] [`*сервер*`...]`

Программа `fetchmail` может забирать почту с серверов, поддерживающих любой протокол приема почты: POP2, POP3, IMAP2bis, IMAP4, IMAP4rev1. Кроме того, она также поддерживает и расширения протокола ESMTP.

Каждое сообщение, принятое `fetchmail`, обычно пересылается по SMTP на порт 25 локальной машины. Далее почта доставляется локально системным транспортным агентом (MTA), в качестве которого обычно используется `sendmail`, но можно применить и программы `smail`, `mmdf`, `exim` и `qmail`. Все механизмы управления доставкой (например, файлы .forward) обрабатываются системным агентом MDA, и после этого срабатывают агенты локальной доставки почты.

Если порт 25 недоступен, а `fetchmail` знает о надежном локальном MDA, то этот MDA и будет использован для локальной доставки. Во время компиляции `fetchmail` обычно пытается найти исполняемые файлы `procmail` и `sendmail`.

Если у вас есть программа `fetchmailconf`, то с ее помощью можно настроить или отредактировать параметры, указанные в файле конфигурации ~/.fetchmailrc. Эта программа работает под управлением системы X Window и требует установленных программ `Python` и `Tk`. Поведение программы `fetchmail` определяется опциями командной строки и конфигурационным файлом ~/.fetchmailrc, причем опции имеют более высокий приоритет по сравнению с параметрами, указанными в файле ~/.fetchmailrc.

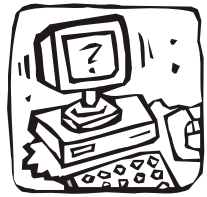
Почти во всех режимах требуется аутентификация клиента. Эта процедура сходна с механизмом, используемым в утилите `ftp`. Проверка правильности идентификатора пользователя и пароля зависит от применяемой на почтовом сервере системы безопасности.

Если почтовый сервер работает на UNIX-системе, в которой вы имеете свою учетную запись, то `fetchmail` будет использовать ваши обычные имя и пароль. Если вы работаете под одним и тем же именем как на сервере, так и на клиенте, то можете не указывать идентификатор пользователя в опции `-u` — по умолчанию используется ваше имя на клиентской машине. Если же на сервере используется другое имя, укажите его в этой опции. Например, если ваше учетное имя localuser на машине с именем localhost, вы можете запустить `fetchmail` следующим образом:

```
fetchmail -u localuser localhost
```

По умолчанию `fetchmail` выполняет интерактивный запрос вашего пароля перед установлением соединения. Это самый безопасный способ применения `fetchmail`, гарантирующий, что пароль не будет расшифрован. Однако можно указать свой пароль и в файле ~/.fetchmailrc. Это полезно при запуске `fetchmail` в режиме демона или в командных файлах.

Если не задан пароль, и программа `fetchmail` не обнаружила его в вашем файле ~/.fetchmailrc, то будет предпринята попытка обнаружить пароль в файле ~/.netrc в домашнем каталоге пользователя. Если это закончится неудачей, то будет выполняться интерактивный запрос. На почтовых серверах, не содержащих обычных учетных записей пользователей, ваши идентификатор и пароль обычно создаются администратором сервера при открытии почтового ящика.



Глава 10

UNIX и Интернет

Как и любая современная операционная система, UNIX обеспечивает широкие возможности для работы пользователя в Интернете. Термином "Интернет" в настоящее время обозначают весьма широкий круг технологий, включающих передачу, прием и обработку данных на компьютерах, входящих как в локальные (местные), так и глобальные сети. Некоторые технологии мы уже рассмотрели (telnet, FTP, электронную почту) в предыдущих главах, сейчас же обратим внимание на один из наиболее распространенных сервисов, благодаря которому Интернет и стал таким популярным, — World Wide Web (WWW). В большинстве случаев, когда речь заходит об Интернете, подразумевают, как правило, именно World Wide Web. Идея WWW появилась у физиков, работающих в центре ядерных исследований CERN, — необходимо было передавать друг другу результаты экспериментов в виде рисунков и диаграмм, а не только текстов. В начале 90-х годов прошлого столетия идея нашла свое воплощение в технологии WWW, а к настоящему времени без Интернета трудно вообще представить жизнь современного человека.

Анализ возможностей, предоставляемых Интернетом, может занять не один десяток, а иногда и не одну сотню страниц, более того, этой тематике посвящена не одна сотня книг. В данной главе мы рассмотрим базис, на котором построено взаимодействие пользователей в Интернете, а именно, протокол HTTP.

10.1. Обмен данными в Интернете

В практическом плане взаимодействие компьютеров (хостов) в Интернете осуществляется, как обычно, по схеме "клиент-сервер". Когда говорят, что пользователь подключается к Интернету, то в подавляющем большинстве случаев подразумевают установку соединения компьютера, на котором работает

пользователь, с одним из серверов, предоставляющих услуги Интернета. Запрашиваемая информация располагается обычно на Web-сервере и в простейшем варианте представляет собой одну или несколько страниц, разработанных посредством языка HTML (HyperText Markup Language). Совокупность таких HTML-страниц, а также других ресурсов (текстовых, мультимедийных файлов, баз данных и т. д.), связанных определенным образом, называют Web-сайтом.

Сначала рассмотрим само определение "Web-сервер". Когда употребляют этот термин, то подразумевают одно из двух:

- компьютер, принимающий запросы HTTP-клиентов, которые называются Web-браузерами, и возвращающий им Web-страницы, представляющие собой HTML-документы и ссылки на другие объекты (рисунки и т. д.);
- программу, обеспечивающую функции, описанные выше.

В этой главе мы будем рассматривать Web-сервер как программу, выполняющую обработку данных по протоколу HTTP и взаимодействующую с несколькими клиентами.

Обмен данными между клиентом и Web-сервером осуществляется по протоколу HTTP (Hypertext Transfer Protocol, протокол передачи гипертекстовых файлов). Этот протокол используется на уровне приложений для распределенных информационных систем и позволяет общаться системам с разной архитектурой посредством передачи HTML-страниц. По этой причине очень часто Web-серверы называют HTTP-серверами, и эти термины используются как синонимы.

В общем случае клиент взаимодействует с Web-сервером посредством специальной программы, называемой браузером (browser) (рис. 10.1).

Для входа в Интернет пользователь запускает браузер (просмотрщик), после чего набирает адрес сетевого ресурса. При успешном соединении Web-сервер возвращает клиенту ответ в виде HTML-страницы. На одном Web-сервере может располагаться одновременно несколько сайтов, а сами Web-серверы в подавляющем большинстве случаев реализованы в операционных системах UNIX. HTTP-сервер работает по протоколу TCP, принимая запросы на порт 80 (по умолчанию).

Каким образом осуществляется взаимодействие клиента и сервера или, что более точно, браузера и Web-сервера? Для понимания этого процесса мы рассмотрим более подробно протокол HTTP. В его основу, как и для других протоколов, положен принцип "запрос — ответ". Обмен по протоколу HTTP осуществляется посредством специальных методов (мы их рассмотрим далее), которые предписывают выполнить последовательность тех или иных

действий. Иными словами, метод можно представить как команду на выполнение определенных действий. Программа-клиент устанавливает связь с обслуживающей программой-получателем (сервером) и посылает запрос серверу в формате, который включает:

- метод запроса;
- универсальный идентификатор ресурсов (Universal Resource Identifier, URI);
- версию протокола, за которой следует MIME-подобное сообщение, содержащее управляющую информацию запроса, информацию о клиенте и, иногда, тело сообщения.

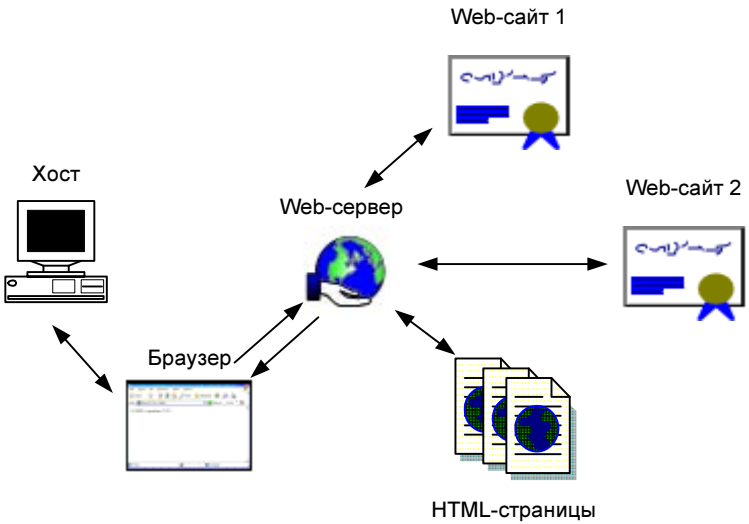


Рис. 10.1. Схема стандартного запроса к Интернету

В ответ сервер посылает клиенту сообщение, содержащее строку статуса (включая версию протокола и код статуса), за которой следует сообщение, схожее по структуре с описанным в спецификации MIME. Данное сообщение включает в себя информацию о сервере, информацию о содержании ответа и само тело ответа. Следует отметить, что одна программа может быть как клиентом, так и сервером.

В общем случае схема HTTP-сеанса выглядит так, как представлено на рис. 10.2.

Обычно запрос клиента представляет собой требование передать HTML-документ или какой-нибудь другой ресурс, а ответ сервера содержит код этого ресурса.

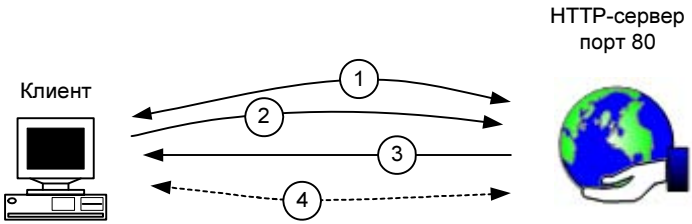


Рис. 10.2. Схема сеанса клиента и HTTP-сервера: 1 — установление соединения; 2 — запрос клиента; 3 — ответ сервера; 4 — разрыв соединения

Во время сеанса по протоколу HTTP промежуточное состояние между парами "запрос — ответ" не сохраняется. Это означает, что обмен сообщениями в каждой паре "запрос — ответ" является независимым от остальных — каждый новый запрос получения ресурса инициирует выполнение отдельных методов и получение ответа.

Для большинства приложений сеанс связи открывается клиентом для каждого запроса и закрывается сервером после окончания ответа на запрос. Тем не менее, это не является особенностью протокола. Как клиент, так и сервер должны иметь возможность закрывать сеанс связи, например, в результате какого-нибудь действия пользователя. В любом случае разрыв связи, инициированный любой стороной, прерывает текущий запрос независимо от его статуса.

На практике используются версии 1.0 и 1.1 данного протокола. Стандарт HTTP предоставляет открытое множество методов, которые могут быть использованы для указания целей запроса. При этом для указания ресурса применяется методика ссылок, реализованная в форме универсального идентификатора ресурсов (Universal Resource Identifier, URI), позволяющего осуществить поиск ресурса либо по местонахождению (Universal Resource Locator, URL), либо по имени (Universal Resource Name, URN).

Рассмотрим подробнее составляющие запроса клиента и ответа сервера. Запрос клиента состоит из:

- ❒ заголовка запроса, в который входят:
  - строка состояния;
  - поле заголовка;
- ❒ пустой строки;
- ❒ тела запроса.

Строка состояния заголовка запроса должна соответствовать следующему формату:

Метод_запроса URL *версия_протокола*_HTTP

Здесь URL однозначно определяет требуемый ресурс, а с помощью *Метода_запроса* задается способ воздействия на указанный ресурс. Версия HTTP-протокола, как правило, задается в формате:

HTTP/*версия.модификация*

Пример указания версии показан далее:

```
HTTP/1.0
```

Остановимся более подробно на методах запроса протокола HTTP. К ним относятся GET, POST, PUT, DELETE и др. В качестве примера рассмотрим методы GET и POST.

Формально метод GET предназначен для получения ресурса с указанным URL. Если в заголовке запроса указан метод GET, то сервер должен включить код указанного ресурса в ответ клиенту. URL может указывать как на HTML-страницу, файл с рисунком или какие-то другие данные, так и на исполняемую программу. В этом случае клиенту возвращаются данные, сгенерированные в процессе выполнения этой программы, которая запускается на выполнение на сервере. Следует отметить, что метод GET может применяться не только с целью получения информации, но и для передачи данных на сервер.

Метод POST служит для передачи данных на сервер, хотя его можно использовать и для получения информации с сервера. Еще одно применение метода POST — запуск исполняемого файла программы на сервере.

Поля заголовка запроса позволяют уточнить запрос, т. е. передать серверу дополнительную информацию. Поле заголовка имеет следующий формат:

Имя_поля: *Значение*

Проанализируем смысл некоторых, наиболее важных полей заголовка HTTP-запроса:

- If-Unmodified-Since — задает условие для выполнения GET-запроса, указывая дату и время предыдущего обращения к ресурсу. Если с указанного времени ресурс не изменялся, то он не будет передан клиенту. В этом случае клиент (браузер) извлекает предыдущую копию документа из кэша;
- Accept — позволяет сообщить серверу MIME-типы данных, обрабатываемых клиентом (например, для конкретизации типов графических файлов, поддерживаемых браузером). Это поле может иметь несколько значений, отделяемых друг от друга запятыми;

- `Accept-Charset` — позволяет задать перечень поддерживаемых наборов символов (кодировок);
- `Connection` — позволяет управлять TCP-соединением. Если в качестве значения этого поля задать `Close`, то после обработки запроса сервер должен разорвать соединение, значение `Keep-Alive` означает, что соединение разрывать не нужно.

Что же касается тела запроса, то в большинстве случаев оно отсутствует. Если же тело запроса не пустое, то используются такие поля заголовка HTTP-запроса, как `Content-Type` и `Content-Length`. В поле `Content-Type` указывается MIME-тип данных, содержащихся в теле запроса, а значение поля `Content-Length` задает число символов, содержащихся в теле запроса.

Ответ сервера похож на запрос и может быть представлен в следующем формате:

- заголовок ответа, состоящий из:
  - строки состояния;
  - поля заголовка;
- пустая строка;
- тело ответа.

Строка состояния имеет следующий формат:

Версия_протокола Код_ответа Комментарий

Версия протокола передается в том же формате, что и в клиентском запросе. Код ответа представляет собой трехзначное десятичное число, в котором зашифрован результат обслуживания запроса. Комментарий расшифровывает код ответа и представляет собой текстовую строку, например:

```
HTTP/1.0 200 OK
```

Поля заголовка HTTP-ответа имеют такую же структуру, как и поля заголовка запроса, причем некоторые поля совпадают. Например, с помощью поля `Content-Type` также указывается MIME-тип данных, содержащихся в теле ответа. С помощью этого поля клиент может определить способ обработки полученных данных.

Тело ответа содержит код ресурса, передаваемого клиенту в ответ на запрос, при этом ресурсом необязательно может быть HTML-файл. В соответствии со спецификацией MIME, для описания формата данных используются тип и подтип в виде *тип*`/`*подтип*, например, `text/plain` (обычный текстовый файл), `text/html` (HTML-документ), `image/gif` (изображение в формате GIF), `image/jpeg` (изображение в формате JPEG), `application/msword` (документ в формате Microsoft Word), `video/avi` (видеофрагмент в формате AVI) и т. д.

Рассмотрим структуру кода ответа. Первая цифра кода ответа определяет класс ответа, а остальные две представляют собой номер ответа внутри класса. В используемых в настоящее время реализациях протокола HTTP первая цифра не может быть больше 5. На данный момент определены следующие классы ответов:

- класс информационных сообщений. Код ответа, начинающийся с 1, означает, что сервер продолжает обработку запроса. Заметим, что при взаимодействии HTTP-клиента и HTTP-сервера сообщения этого класса используются редко;
- успешная обработка запроса клиента;
- перенаправление запроса. Чтобы запрос был обслужен, необходимо предпринять дополнительные действия;
- ошибка клиента (например, синтаксическая ошибка в запросе);
- ошибка сервера (по тем или иным причинам сервер не в состоянии выполнить запрос).

Примеры некоторых кодов ответа сервера приведены в табл. 10.1.

Таблица 10.1. *Коды ответов сервера*

Код ответа	Пояснительное сообщение	Комментарии
100	Continue	Часть запроса принята, и сервер ожидает от клиента продолжения запроса
200	OK	Запрос успешно обработан, и в ответе клиенту передаются данные, указанные в запросе
301	Multiple Choice	Запрос указывает более чем на один ресурс
400	Bad Request	В запросе клиента обнаружена синтаксическая ошибка
403	Forbidden	Имеющийся на сервере ресурс недоступен для данного пользователя
404	Not Found	Ресурс, указанный клиентом, на сервере отсутствует
405	Method Not Allowed	Сервер не поддерживает метод, указанный в запросе
500	Internal Server Error	Один из компонентов сервера работает некорректно
501	Not Implemented	Функциональных возможностей сервера недостаточно, чтобы выполнить запрос клиента
503	Service Unavailable	Служба временно недоступна
505	HTTP Version not Supported	Версия HTTP, указанная в запросе, не поддерживается сервером

Примечание

Полное описание протокола HTTP, а также всех кодов ответов HTTP-сервера приведено в спецификации RFC 2616.

Как уже упоминалось, для выполнения запроса к серверу используется программа, имеющая название браузер (browser). В настоящее время существует достаточно много программ данного класса, предназначенных для работы в операционных системах UNIX (Netscape, Mozilla, Lynx и т. д.). Несмотря на различие в деталях реализации, функционально эти браузеры похожи друг на друга. Это означает, что если вы хотите перейти с одного браузера на другой, то никаких сложностей, как правило, у вас не возникнет.

Функционирование браузера основано на использовании так называемого гипертекста (hypertext). В основе языка гипертекста лежит концепция ссылки (link), о которой было упомянуто ранее. Ссылки выделяются либо цветом (обычно синим), либо подчеркиванием (например, как здесь) и указывают на определенный ресурс (например, на отдельный документ или на его часть). Если щелкнуть мышью на ссылке, то пользователь попадает на указанный ресурс (если, конечно, ссылка является правильной). Для определения названия ресурса, на который указывает ссылка, достаточно поместить курсор мыши поверх ссылки.

Ссылки формируются на основе URL-адресов, представляющих собой определенную комбинацию IP-адреса компьютера и пути к ресурсу, находящемуся в его файловой системе.

Вот примеры URL-адресов:

- **http://csep10.phys.utk.edu/webcourse/browser/textfile.html** — ссылка на HTML-страницу;
- **http://www.techcorps.org.org/webcourse/browser/usa2.gif** — ссылка на рисунок;
- **http://www.techcorps.org/webcourse/browser/goldgate.mpg** — ссылка на видеофайл.

Для работы в Интернете необходимо соответствующим образом настроить аппаратную часть компьютера и программное обеспечение. Процедуры настройки доступа в Интернет в разных реализациях операционной системы UNIX приблизительно одинаковы и включают несколько шагов:

1. Настройка коммуникационного оборудования (модема, сетевого устройства и т. д.). На этом шаге требуется установить драйвер устройства (если он еще не установлен) и указать параметры обмена данными (тип обмена, скорость, опции инициализации и т. д.). Например, при использовании

модема для подключения к Интернету нужно, как правило, указать скорость обмена, наличие/отсутствие контроля четности, способы синхронизации. Если система при инсталляции установила драйвер модема, то лучше при первоначальной настройке никаких изменений не проводить, а сделать это позже, когда соединение будет гарантированно работать. Если подключение к Интернету осуществляется посредством сетевой карты, то ее нужно настроить с помощью программы `ifconfig` или из графической консоли, выбрав соответствующие опции.

2. Настройка доступа к серверу провайдера услуг Интернета. В большинстве случаев сервер провайдера выполняет автоматическое конфигурирование клиента (присвоение динамического IP-адреса, установку атрибутов безопасности и т. д.), так что от клиента требуется минимум настроек. Если, например, подключение осуществляется посредством модема, то обычно необходимо указать номер дозвона, имя и/или IP-адрес сервера Интернета, а также атрибуты учетной записи клиента (имя и пароль). Если же используется соединение через сетевой интерфейс, то номер дозвона не понадобится. Более сложная настройка понадобится, если используется тунеллирование или иные способы повышения безопасности соединения.

10.2. Простейший Web-сервер

В данном разделе мы рассмотрим практические аспекты взаимодействия клиентов Интернета с Web-серверами по протоколу HTTP и, кроме этого, проведем анализ программного кода простейшего HTTP-сервера. Материал, представленный далее, окажет существенную помощь как начинающим разработчикам, желающим заняться созданием собственных Web-серверов, так и широкому кругу пользователей, создающих Web-страницы и Web-сайты.

Несмотря на некий ореол таинственности, окружающий процесс разработки Web-серверов, на самом деле написать простой HTTP-сервер не так уж и сложно. Для этого нужно знать принципы функционирования TCP-серверов и уметь запрограммировать простое сетевое приложение. Мы уже рассматривали примеры сетевых приложений в *главе 8*, поэтому многие аспекты функционирования HTTP-сервера читателям будут понятны.

Рассмотрим простейший HTTP-сервер, который в ответ на запрос клиента (Web-браузера) пересылает ему HTML-страницу.

Исходный текст программы (она называется `httpsrv_1`), в которой реализован Web-сервер, показан далее:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

#define port 8081

int fdsock, new_con;
fd_set socks;
char *page =
"HTTP/1.1 200 OK\n"
"Content-type: text/html\n"
"\n"
"<html><body><h1 align=\"center\"> DEMO Http-Server START Page
</h1></body></html>\n";

int main(void)
{
 struct sockaddr_in server;
 int pid, ret;
 char buf[4096];

 bzero(&server, sizeof(server));
 server.sin_addr.s_addr = htonl(INADDR_ANY);
 server.sin_family = AF_INET;
 server.sin_port = htons(port);

 fdsock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
 bind(fdsock, (struct sockaddr*)&server, sizeof(server));
 listen(fdsock, 5);

 printf("Waiting for HTTP-request on port 8081...\n");
 while(1)
 {
  new_con = accept(fdsock, 0, 0);
  if (new_con > 0)
   {
```

```
    pid = fork();
    if (pid == 0)
     {
      close(fdsock);
      ret = recv(new_con, buf, sizeof(buf), 0);
      if (ret > 0)
       {
        buf[ret] = 0;
        printf("REQUEST: %s\n", buf);
        send(new_con, page, strlen(page), 0);
       }
      close(new_con);
      exit(0);
     }
    close(new_con);
   }
 }
 close(fdsock);
 return 0;
}
```

Проанализируем исходный текст. Здесь вместо стандартного TCP-порта с номером 80 используется порт 8081. Это принципиально ничего не меняет, поскольку мы вправе для наших сетевых приложений выбирать любой порт из диапазона допустимых значений, но исключает конфликт портов, если в вашей операционной системе уже работает стандартный Web-сервер, использующий порт 80.

Для упрощения изложения исходный текст HTML-страницы размещен прямо в исходном тексте приложения (переменная `page` выделена жирным шрифтом) и представляет собой строку текста:

```
char *page =
"HTTP/1.1 200 OK\n"
"Content-type: text/html\n"
"\n"
"<html><body><h1 align=\"center\"> DEMO Http-Server START Page
</h1></body></html>\n";
```

Операторы

```
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_family = AF_INET;
```

```
server.sin_port = htons(port);

fdsock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
bind(fdsock, (struct sockaddr*)&server, sizeof(server));
listen(fdsock, 5);
```

выполняют инициализацию структуры `server`, создают сокет с дескриптором `fdsock` (функция `socket()`), затем, после привязки `fdsock` к структуре `server` (функция `bind()`), устанавливают сокет в режим прослушивания на порту 8081 (функция `listen()`).

Обработка входящих запросов клиентов выполняется в бесконечном цикле `while`. Если новое соединение успешно установлено (функция `accept()`), то для обмена данными по этому соединению создается новый процесс (системный вызов `fork()`):

```
pid = fork();
if (pid == 0)
 {
  close(fdsock);
  ret = recv(new_con, buf, sizeof(buf), 0);
  if (ret > 0)
   {
    buf[ret] = 0;
    printf("REQUEST: %s\n", buf);
    send(new_con, page, strlen(page), 0);
   }
  close(new_con);
  exit(0);
. . .
```

Использование дочернего процесса позволяет серверу обрабатывать входящие запросы независимо друг от друга, что очень важно для поддержания оптимального уровня производительности и позволяет исключить блокировки запросов друг другом.

Остановимся более подробно на фрагменте кода, реализующем дочерний процесс. Напомню, что при создании дочернего процесса ему передаются копии дескрипторов открытых файлов родительского процесса, в данном случае `fdsock` и `new_con`. Дескриптор `fdsock` связан с прослушивающим сокетом сервера и в дочернем процессе не используется, поэтому его можно закрыть без каких-либо последствий для родительского процесса, что и делает системный вызов `close(fdsock)`. Затем, по завершении обмена данными, следует закрыть и рабочее соединение (системный вызов `close(new_con)`).

Кроме того, порожденный процесс должен заканчиваться либо функцией `exit()`, либо `return`.

Обмен данными между клиентом и сервером осуществляется в дочернем процессе посредством двух операторов:

```
. . .
int ret = recv(new_con, buf, sizeof(buf), 0);
send(new_con, page, strlen(page), 0);
. . .
```

С помощью функции `recv()` принимается запрос от клиента, который затем отображается на экране консоли. Функция `send()` позволяет отправить HTML-страницу клиенту. Следует отметить, что сервер, кроме передачи самой страницы, может дополнительно передавать MIME-заголовок, указывая при этом статус сообщения и тип возвращаемого документа. В нашем случае (см. объявление переменной `page`) возвращается следующая информация:

```
"HTTP/1.1 200 OK\n"
"Content-type: text/html\n"
```

Для простоты изложения в данном программном коде не производится анализ количества одновременно работающих подключений (не более 5). Читатели при желании смогут доработать исходный текст и выполнить такую проверку, а также возможную реакцию на превышение количества соединений (например, засыпание HTTP-сервера в ожидании закрытия соединений).

Для большей наглядности работу демонстрационного HTTP-сервера можно представить на рис. 10.3.

Предположим, что с HTTP-сервером взаимодействуют 3 клиента. Из рисунка видно, что клиент 1, клиент 2 и клиент 3 посылают запросы (1)—(3) HTTP-серверу. Для обмена информацией с клиентами сервер создает три дочерних процесса (1-1)—(3-3) посредством системного вызова `fork()` для обработки входящих соединений. При этом основной процесс сервера продолжает слушать порт 8081 в ожидании новых запросов на обслуживание.

В данном примере программный код сервера не выполняет анализ (parsing) HTTP-запроса, поступающего от клиента, что обычно требуется при создании более сложных серверных приложений.

Обратите внимание, что HTTP-сервер можно запустить на локальной машине с привязкой к сетевому интерфейсу обратной связи (IP-адрес 127.0.0.1).

Для проверки работы сервера следует запустить откомпилированный код из командной строки:

```
bash-3.00# ./httpsrv_1
Waiting for HTTP-request on port 8081...
```

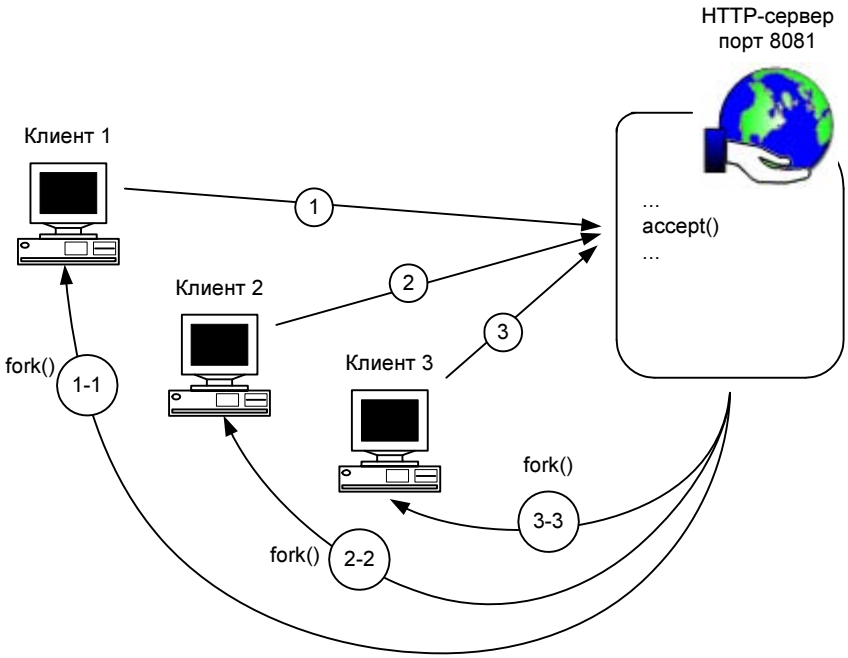


Рис. 10.3. Схема работы демонстрационного HTTP-сервера

Затем в поле адреса браузера ввести строку:

```
http://127.0.0.1:8081
```

После этого в окне браузера появится содержимое HTML-страницы (рис. 10.4). Здесь показана работа HTTP-сервера в операционной системе Linux Red Hat 9, а запросы к серверу осуществляются из Web-браузеров Mozilla и Konqueror.

На экране консоли, с которой запущен HTTP-сервер, будет отображен запрос браузера к серверу:

```
# ./httpsrv_1
Waiting for HTTP-request on port 8081...
REQUEST: GET / HTTP/1.1
Host: 127.0.0.1:8081
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.2.1) Gecko/20030225
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,text/css,*/*;q=0.1
```

```
Accept-Language: en-us, en;q=0.50
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
```

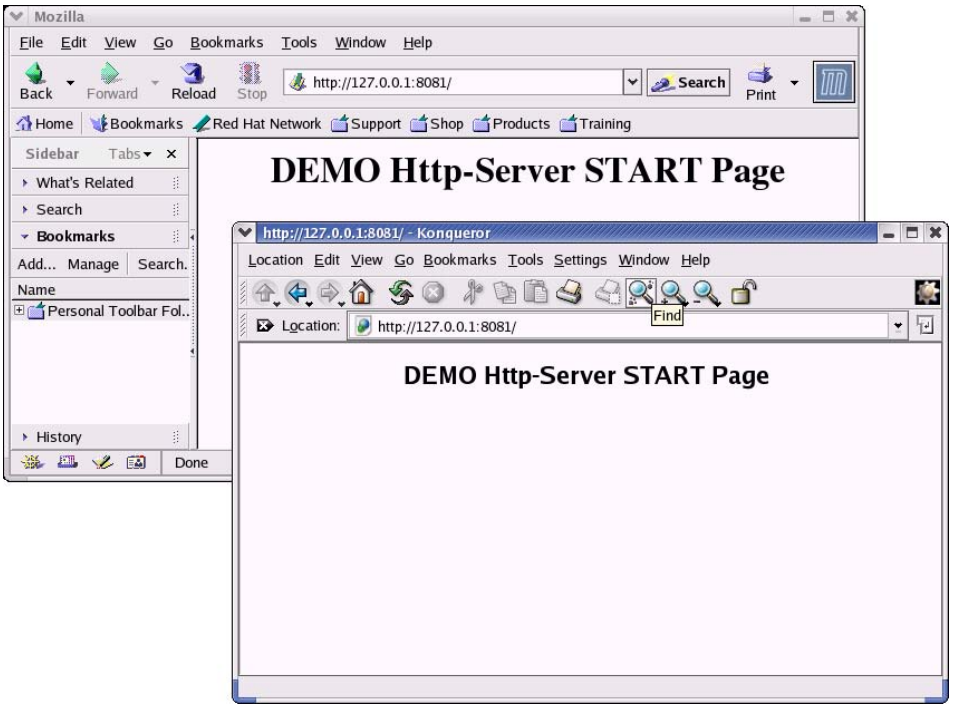


Рис. 10.4. Отображение HTML-страницы в окне браузера

По выводимой информации легко проследить структуру запроса и смысл отображаемых полей (они были описаны ранее в данном разделе).

Несмотря на свою простоту, приведенный пример реализации HTTP-сервера является эффективным решением при организации взаимодействия с несколькими клиентами одновременно, и на его основе можно построить довольно сложные и многофункциональные Web-серверы. Более того, алгоритм ветвления основного процесса для обработки входящих соединений используется большинством разработчиков программного обеспечения при создании Web-серверов.

Естественно, что возможны и другие варианты реализации обработки запросов Web-сервера, например, посредством библиотечных функций `select()` или `poll()`.

Функционирование демонстрационного сервера можно проверить и другим способом — написать простую программу-клиент, получающую от сервера текст HTML-страницы. В этом примере, и это очень важно, показано, как формируется HTTP-запрос к Web-серверу, что позволяет понять принципы функционирования таких сложных программ, какими являются Web-браузеры.

Исходный текст HTTP-клиента (программа называется `httpclient_1`) показан далее:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#define port 8081

int main(int argc, char* argv[])
{
 if (argc != 2)
  {
   printf("Usage: %s hostname\n", argv[0]);
   exit(1);
  }

 char *req = "GET / HTTP/1.1\n";
 struct sockaddr_in server;
 struct hostent *host;
 int fdsock, new_con;
 char buf[4096];
 int bytesRead = 0;

 bzero(&server, sizeof(server));
 bzero(&buf, sizeof(buf));

 host = gethostbyname(argv[1]);
 memcpy(&server.sin_addr, host->h_addr_list[0], host->h_length);
 server.sin_family = AF_INET;
```

```
 server.sin_port = htons(port);

 fdsock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
 connect(fdsock, (struct sockaddr*)&server, sizeof(server));
 send(fdsock, req, strlen(req), 0);
 while(1)
 {
  bytesRead = recv(fdsock, buf, sizeof(buf), 0);
  if (bytesRead == 0)
    break;
  printf("%s", buf);
 }
 close(fdsock);
 return 0;
}
```

Прежде чем анализировать исходный текст программы-клиента, остановлюсь на принципах формирования HTTP-запроса в данном приложении, поскольку это самая сложная его часть.

При формировании запроса очень важно правильно интерпретировать URL. Унифицированные локаторы ресурсов протокола HTTP можно представить в общем формате следующим образом:

`http://`*`имя_хоста`*`:`*`порт`*`/`*`путь_к_документу`*`#`*`фрагмент_документа`*

Все URL протокола HTTP начинаются с префикса **http://**. Далее следует имя хоста, например, **www.linux.org**, двоеточие и номер порта, через который Web-сервер принимает входящие запросы. Двоеточие и номер порта могут быть опущены, при этом подразумевается, что Web-сервер работает на стандартном порту 80, что верно в подавляющем большинстве случаев. За именем хоста и номером порта указывается путь к требуемому документу, для определения которого используются соглашения по обозначению пути к файлам, принятые в UNIX-системах. Далее следуют символ # и имя фрагмента, обозначающее подраздел документа, к которому нужно обратиться.

В данной программе интерпретация URL выполняется оператором

```
char *req = "GET / HTTP/1.1\n";
```

При этом URL преобразуется в форму

```
GET /путь_к_документу HTTP/1.0\n
```

В строку запроса может включаться информация, позволяющая серверу определить, как взаимодействовать с клиентом. Метод `GET` принимает два парамет-

ра: собственно запрос и используемый протокол. Сервер может анализировать запрос, выделяя имя каталога и имя документа. В общем случае запросы могут быть намного сложнее, чем в нашем примере. Протокол HTTP 1.0 допускает наличие пробелов в путевом имени, поэтому запрос включает в себя все, что находится между начальным символом косой черты и строкой `HTTP/`.

Отправка запроса HTTP-серверу выполняется при помощи оператора

```
send(fdsock, req, strlen(req), 0);
```

Далее программа ожидает ответа от сервера и выводит полученную информацию на экран консоли:

```
while(1)
 {
  bytesRead = recv(fdsock, buf, sizeof(buf), 0);
  if (bytesRead == 0)
    break;
  printf("%s", buf);
 }
```

Откомпилируем исходный текст программы-клиента и запустим HTTP-сервер и клиент на выполнение. Вот как может выглядеть взаимодействие клиента и сервера в операционной системе Linux:

- сервер:

```
# ./httpsrv_1
Waiting for HTTP-request on port 8081...
REQUEST: GET / HTTP/1.1
```

- клиент:

```
# ./httpclient_1 localhost
HTTP/1.1 200 OK
Content-type: text/html

<html><body><h1 align="center"> DEMO Http-Server START Page
</h1></body></html>
```

Ответ клиенту состоит из двух частей: заголовка, содержащего информацию о возвращаемом документе, и самого затребованного документа. Подобная программа, если ее усовершенствовать, в ряде случаев может оказать существенную помощь при отладке собственного Web-сайта или для проверки работоспособности Web-сервера.

10.3. Web-сервер Apache

В настоящее время многие занимаются разработкой собственных Web-страниц и даже Web-сайтов. Это обусловлено тем, что Интернет, как бурно развивающаяся отрасль, перестал быть просто источником получения информации посредством HTML-страниц, а предоставляет с каждым днем все бóльшие и бóльшие возможности для образования, видеоконференций, электронной коммерции, работы по удаленному доступу и т. д.

Разработку собственного Web-сайта практически невозможно выполнить, не имея средства отладки, в качестве которого может выступать один из популярных (и бесплатных) Web-серверов. Кроме того, выбор Web-сервера чрезвычайно важен для провайдеров услуг Интернета. Современный рынок программных продуктов предлагает много разработок Web-серверов. Наиболее популярным и широко используемым среди них является Web-сервер Apache. Он обладает целым рядом преимуществ по сравнению с остальными:

- ❒ программа является бесплатной;
- ❒ программа хорошо себя зарекомендовала;
- ❒ распространяется в виде исходного кода или уже готовых модулей;
- ❒ программа имеет хорошую техническую поддержку;
- ❒ все обновления выполняются очень быстро.

Для установки Apache в операционных системах UNIX обычно компилируют исходные тексты программы, предварительно загруженные с сайта **www.apache.org** или из других источников. Перед компиляцией исходных текстов специальная программа `configure` анализирует конфигурацию конкретной системы, что позволяет исключить из программного кода ненужные части.

Далее с помощью команды `make` запускается компиляция программы. При успешной компиляции создается исполняемый файл с именем `httpd`, который является запускающим для Web-сервера. Для дополнительных настроек сервера можно использовать несколько файлов конфигурации, важнейшим из которых является httpd.conf.

Остановимся более подробно на инсталляции и настройке Web-сервера Apache на примере операционной системы Linux. Напомню, что дистрибутив Apache можно загрузить с Web-сайта **www.apache.org**.

Последней версией Apache на момент написания книги являлась версия 2.2.2, поэтому все дальнейшие рассуждения касаются именно этой версии. Web-сервер Apache является настолько популярным, что его включают во многие дистрибутивы операционных систем UNIX, поэтому не исключен вариант,

что в вашей системе уже установлена одна из более ранних версий Apache. Перед установкой Web-сервера желательно проверить, выполняется ли процесс `htttpd`, что легко сделать посредством команды `ps -ef|grep htttpd`. Если результат выполнения команды напоминает тот, что показан далее, то это означает, что Web-сервер уже запущен в вашей системе:

```
# ps -ef|grep httpd
  root   512   1    0  13:42:47  ?       0:00 /usr/apache/bin/httpd
  nobody 514   512  0  13:42:48  ?       0:00 /usr/apache/bin/httpd
  nobody 513   512  0  13:42:48  ?       0:00 /usr/apache/bin/httpd
  nobody 515   512  0  13:42:48  ?       0:00 /usr/apache/bin/httpd
  nobody 517   512  0  13:42:48  ?       0:00 /usr/apache/bin/httpd
  nobody 516   512  0  13:42:48  ?       0:00 /usr/apache/bin/httpd
    root 716   710  0  13:46:21  pts/4   0:00 grep httpd
```

В данном случае выполняется одна из предыдущих версий Apache, поскольку демон `httpd` для версий 2.2.*x* запускается из каталога /usr/local/apache2/bin. Процесс-демон `httpd` выполняется как сетевой сервер с предварительным ветвлением (preforking), а это означает, что при запуске основной процесс-демон (PPID = 1) с идентификатором PID, равным 512, запускает пять дочерних процессов с идентификаторами PID от 513 до 517, ожидающих подключения клиентов. По такой схеме запускаются все версии Web-сервера Apache. Естественно, что для вашей системы все значения идентификаторов процессов PID будут другими.

Для установки новой версии Web-сервера не обязательно удалять работающую версию Apache — можно выполнить инсталляцию в другой каталог, отличный от рабочего (в данном случае рабочим каталогом выполняющегося сервера является /usr/apache). При этом нужно тщательно изучить документацию на устанавливаемый и работающий серверы, чтобы не затереть файлы конфигурации текущей версии Apache. Еще один вариант — выполнить апгрейд (upgrade) предыдущей версии (если это, конечно, возможно). Ввиду ограниченности объема книги рассмотреть все варианты установки Web-сервера Apache не представляется возможным, поэтому далее мы будем предполагать, что в операционной системе UNIX Web-сервер не установлен, и выполним установку с "нуля".

Итак, сохраним дистрибутив в каком-нибудь каталоге, например, DIST. Файл дистрибутива обычно представлен в виде архива с именем httpd-*NN*.tar.gz, где *NN* — номер версии и имеет размер несколько мегабайтов:

```
# ls -l DIST
total 12832
```

```
-r-xr-xr-x   1 root     sys       322474 Mar 19 22:19 apache_httpd.pdf
-r-xr-xr-x   1 root     sys      6224116 Mar 19 22:15 httpd-2.2.2.tar.gz
```

Напомню, что в данном случае будет проинсталлирована версия 2.2.2 Web-сервера Apache. Следующий шаг — разархивация полученного дистрибутива, для чего нужно последовательно выполнить команды `gzip` и `tar`:

```
# gzip -d httpd-2.2.2.tar.gz
# ls -l
total 60288
-r-xr-xr-x   1 root     sys       322474 Mar 19 22:19 apache_httd.pdf
-r-xr-xr-x   1 root     sys      30504960 Mar 19 22:15 httpd-2.2.2.tar
# tar xvf httpd-2.2.2.tar
# ls -l
total 60290
-r-xr-xr-x   1 root     sys       322474 Mar 19 22:19 apache_httpd.pdf
drwxr-xr-x  11 1000     1000        1024 Nov 29 08:19 httpd-2.2.2
-r-xr-xr-x   1 root     sys      30504960 Mar 19 22:15 httpd-2.2.2.tar
```

В результате выполнения данных команд в текущем каталоге (в данном случае DIST) будет создан каталог httpd-2.2.2, содержащий исходные тексты программы.

Перед выполнением следующего шага желательно проверить некоторые настройки операционной системы. В частности, нужно выяснить, содержит ли переменная окружения `PATH` пути к каталогам, где находятся компиляторы C (`gcc`, `cc`). Дело в том, что процедура инсталляции Web-сервера вызывает компилятор C для обработки исходных текстов и сборки приложения. Инсталляционная программа попытается найти путь к компиляторам на основании системных настроек, и если поиск окажется неудачным, то инсталляция будет прервана. Самое первое, что нужно в этом случае проверить, — установлен ли вообще пакет разработки на языке C/C++ в системе, и если это не так, установить его. Если данный пакет установлен, то следует прописать путь к компилятору `gcc` в переменной `PATH`, после чего можно начинать инсталляцию Apache.

Простейший способ проверить доступность программы-компилятора — попытаться запустить программу `gcc` или `cc` из текущего каталога без параметров. Если будет получен результат наподобие

```
# gcc
bash: gcc: command not found
```

то это означает, что система не может определить путь к файлу. Задайте команду

```
# find /usr -name "gcc" -print
```

и посмотрите на результат. Если ничего не выведено на экран консоли, то, скорее всего, пакет C/C++ не установлен. Если требуемые файлы обнаружены, то нужно добавить путь к ним в переменную PATH.

Далее следует проверить наличие в операционной системе утилиты make, собственно, выполняющей установку дистрибутива Apache. Если эта программа по каким-либо причинам отсутствует, дальнейшая установка невозможна. Если же make удается обнаружить командой поиска find, то нужно прописать к ней путь через переменную окружения PATH, после чего установку Web-сервера можно продолжить.

На следующем шаге инсталляции следует перейти в каталог httpd-2.2.2 и выполнить программу configure:

```
# cd httpd-2.2.2
# ./configure
```

Программа configure выполняет предварительные настройки и использует для этого каталог /usr/local/apache2, который будет корневым для Web-сервера. Если программа configure завершилась без ошибок, запустите следующие две команды:

```
# make
# make install
```

При отсутствии ошибок можно запустить Web-сервер Apache:

```
# /usr/local/apache2/bin/apachectl -k start
```

Для того чтобы убедиться в работоспособности Apache, вначале проверим, выполняется ли сам процесс. Напомню, что Web-сервер Apache реализован как демон httpd с предварительным ветвлением на пять дочерних процессов (настройка по умолчанию), поэтому выполнение команды ps -ef|grep httpd даст такой результат:

```
# ps -ef|grep httpd
root  28221  1  0 02:10 ?  00:00:00 /usr/local/apache2/bin/httpd -k start
daemon   28222 28221  0 02:10 ?        00:00:00 [httpd]
daemon   28223 28221  0 02:10 ?        00:00:00 [httpd]
daemon   28224 28221  0 02:10 ?        00:00:00 [httpd]
daemon   28225 28221  0 02:10 ?        00:00:00 [httpd]
daemon   28226 28221  0 02:10 ?        00:00:00 [httpd]
root     28228  4864  0 02:11 pts/0    00:00:00 grep httpd
```

Теперь можно попробовать зайти на Web-сервер Apache из какого-нибудь браузера, набрав в строке адреса http://localhost. В ответ получим от сер-

вера HTML-страницу с текстом "It Works". Вид окна браузера Mozilla в операционной системе Red Hat Linux представлен на рис. 10.5.

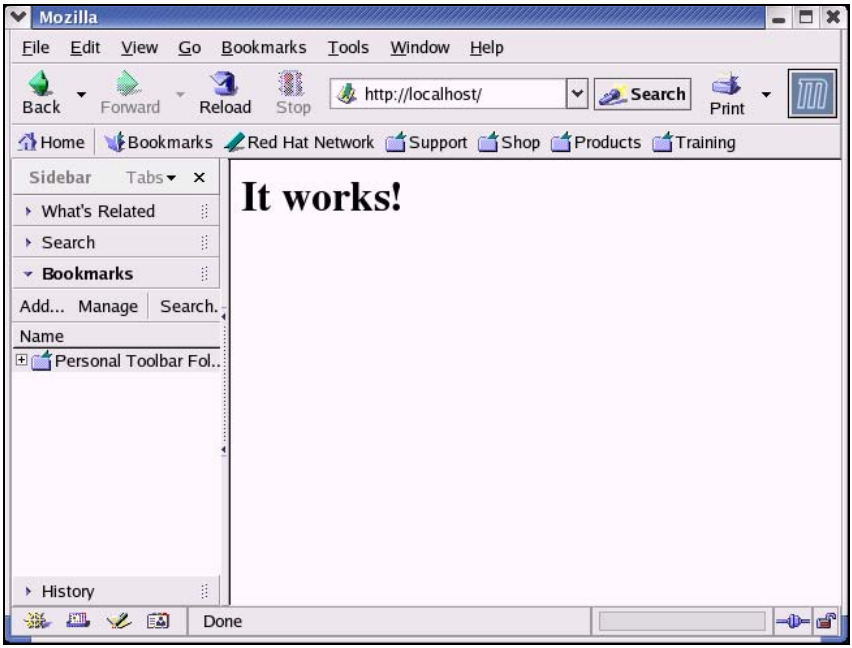


Рис. 10.5. Тестовая страница для Web-сервера Apache

Для остановки Web-сервера следует выполнить команду:

```
# /usr/local/apache2/bin/apachectl -k stop
```

Перезапуск сервера можно выполнить при помощи команды

```
# /usr/local/apache2/bin/apachectl -k restart
```

Проверить номер версии Web-сервера Apache можно при помощи следующей команды:

```
# /usr/local/apache2/bin/apachectl -v
Server version: Apache/2.2.2
Server built:   May  4 2006 18:52:55
```

Полный набор опций программы можно просмотреть, выполнив командный файл `apachectl` без параметров:

```
# /usr/local/apache2/bin/apachectl
Usage: /usr/local/apache2/bin/httpd [-D name] [-d directory] [-f file]
                                    [-C "directive"] [-c "directive"]
```

```
                                    [-k start|restart|graceful|graceful-
                                    stop|stop]
                                    [-v] [-V] [-h] [-l] [-L] [-t] [-S]
```

Смысл опций командного файла `apachectl` достаточно хорошо описан в документации на Web-сервер, поэтому останавливаться на этом мы не будем.

Рассмотрим более подробно настройки сервера. Напомню, что корневым каталогом Web-сервера Apache версий 2.2.*x* является /usr/local/apache2, а большинство важных настроек можно выполнить, отредактировав файл httpd.conf, находящийся в каталоге /usr/local/apache2/conf. Несомненно, остальные настройки являются не менее важными, но при первом знакомстве с сервером их можно оставить без изменения. Вот пример записей файла конфигурации httpd.conf (в сокращении):

```
#
# This is the main Apache HTTP server configuration file.  It contains
# the configuration directives that give the server its instructions.
# See <URL:http://httpd.apache.org/docs/2.2> for detailed information.
# In particular, see
# <URL:http://httpd.apache.org/docs/2.2/mod/directives.html>
# for a discussion of each configuration directive.
#
# Do NOT simply read the instructions in here without understanding
# what they do.  They're here only as hints or reminders.  If you are
# unsure consult the online docs. You have been warned.
#
# Configuration and logfile names: If the filenames you specify for many
# of the server's control files begin with "/" (or "drive:/" for Win32),
# the server will use that explicit path.  If the filenames do *not*
# begin with "/", the value of ServerRoot is prepended - so
# "logs/foo.log"
# with ServerRoot set to "/usr/local/apache2" will be interpreted by the
# server as "/usr/local/apache2/logs/foo.log".
#
# ServerRoot: The top of the directory tree under which the server's
# configuration, error, and log files are kept.
#
# Do not add a slash at the end of the directory path.  If you point
# ServerRoot at a non-local disk, be sure to point the LockFile directive
# at a local disk.  If you wish to share the same ServerRoot for multiple
```

```
# httpd daemons, you will need to change at least LockFile and PidFile.
#
ServerRoot "/usr/local/apache2"
#
# Listen: Allows you to bind Apache to specific IP addresses and/or
# ports, instead of the default. See also the <VirtualHost>
# directive.
#
# Change this to Listen on specific IP addresses as shown below to
# prevent Apache from glomming onto all bound IP addresses.
#
#Listen 12.34.56.78:80
Listen 81
#
# Dynamic Shared Object (DSO) Support
#
# To be able to use the functionality of a module which was built as a
# DSO you have to place corresponding `LoadModule' lines at this location
# so the directives contained in it are actually available _before_ they
# are used.
# Statically compiled modules (those listed by `httpd -l') do not need
# to be loaded here.
#
# Example:
# LoadModule foo_module modules/mod_foo.so
#
#
# If you wish httpd to run as a different user or group, you must run
# httpd as root initially and it will switch.
#
# User/Group: The name (or #number) of the user/group to run httpd as.
# It is usually good practice to create a dedicated user and group for
# running httpd, as with most system services.
#
User daemon
Group daemon
# 'Main' server configuration
#
# The directives in this section set up the values used by the 'main'
```

```
# server, which responds to any requests that aren't handled by a
# <VirtualHost> definition.  These values also provide defaults for
# any <VirtualHost> containers you may define later in the file.
#
# All of these directives may appear inside <VirtualHost> containers,
# in which case these default settings will be overridden for the
# virtual host being defined.
#
#
# ServerAdmin: Your address, where problems with the server should be
# e-mailed.  This address appears on some server-generated pages, such
# as error documents.  e.g. admin@your-domain.com
#
ServerAdmin you@example.com
#
# ServerName gives the name and port that the server uses to identify
# itself. This can often be determined automatically, but we recommend
# you specify it explicitly to prevent problems during startup.
# If your host doesn't have a registered DNS name, enter its IP address
# here.
#
#ServerName www.example.com:80
#
# DocumentRoot: The directory out of which you will serve your
# documents. By default, all requests are taken from this directory, but
# symbolic links and aliases may be used to point to other locations.
#
DocumentRoot "/usr/local/apache2/htdocs"
#
# Each directory to which Apache has access can be configured with
# respect to which services and features are allowed and/or disabled in
# that directory (and its subdirectories).
#
# First, we configure the "default" to be a very restrictive set of
# features.
#
<Directory />
    Options FollowSymLinks
```

```
    AllowOverride None
    Order deny,allow
    Deny from all
</Directory>
 . . .
 . . .
#
# Note that from this point forward you must specifically allow
# particular features to be enabled - so if something's not working as
# you might expect, make sure that you have specifically enabled it
# below.
#
 . . .
 . . .
#
# This should be changed to whatever you set DocumentRoot to.
#
<Directory "/usr/local/apache2/htdocs">
#
# Possible values for the Options directive are "None", "All",
# or any combination of:
# Indexes Includes FollowSymLinks SymLinksifOwnerMatch ExecCGI MultiViews
#
# Note that "MultiViews" must be named *explicitly* --- "Options All"
# doesn't give it to you.
#
# The Options directive is both complicated and important.  Please see
# http://httpd.apache.org/docs/2.2/mod/core.html#options for more
# information.
#
 Options Indexes FollowSymLinks

#
# AllowOverride controls what directives may be placed in .htaccess
# files.
# It can be "All", "None", or any combination of the keywords:
#   Options FileInfo AuthConfig Limit
#
```

```
 AllowOverride None

#
# Controls who can get stuff from this server.
#
Order allow,deny
Allow from all

</Directory>

#
# DirectoryIndex: sets the file that Apache will serve if a directory
# is requested.
#
# The following lines prevent .htaccess and .htpasswd files from being
# viewed by Web clients.
#
<FilesMatch "^\.ht">
    Order allow,deny
    Deny from all
    Satisfy All
</FilesMatch>
#
 . . .
 . . .
#
# DefaultType: the default MIME type the server will use for a document
# if it cannot otherwise determine one, such as from filename extensions.
# If your server contains mostly text or HTML documents, "text/plain" is
# a good value.  If most of your content is binary, such as applications
# or images, you may want to use "application/octet-stream" instead to
# keep browsers from trying to display binary files as though they are
# text.
#
DefaultType text/plain

 . . .
 . . .
#
```

Смысл некоторых важных директив для настройки (выделены жирным шрифтом в листинге) таков:

- `ServerRoot` — указывает каталог, являющийся вершиной дерева подкаталогов, определяющих конфигурацию сервера;
- `Listen` — определяет сетевые настройки (IP-адрес:порт) сервера. Принимается по умолчанию равным 80, хотя можно установить и другое значение, если необходимо работать с несколькими Web-серверами или с разными версиями одного и того же сервера. В данном случае для прослушивающего порта выбрано значение 81;
- `ServerAdmin` — указывает адрес электронной почты, по которому следует отправлять сообщения при возникновении проблем с сервером;
- `ServerName` — указывает имя сервера и порт, на котором он работает. Если невозможно указать имя (например, проблемы с DNS), следует указать IP-адрес сервера;
- `DocumentRoot` — определяет корневой каталог, в котором находятся обрабатываемые ресурсы. При инициализации корневым каталогом становится /usr/local/apache2/htdocs. Для этого каталога следует установить права доступа с помощью директивы `Directory`;
- `FilesMatch` — устанавливает права доступа для файлов с указанным шаблоном. При инициализации Web-сервера запрещается доступ клиентов к файлам .htaccess и .htpasswd;
- `DefaultType` — определяет тип данных, обрабатываемых сервером. Для серверов, обрабатывающих, в основном, HTML или текстовые документы, лучше выбрать значение `text/plain`. Если бóльшая часть информации представляет собой двоичные файлы, например, мультимедиа и/или графику, то следует выбрать значение `application/octet-stream`.

Web-сервер Apache обладает еще одной важной особенностью — он позволяет создавать так называемые виртуальные хосты, используемые для размещения более чем одного Web-сайта на одной машине. Виртуальные хосты привязываются либо к отдельному IP-адресу, либо к имени, позволяя оперировать несколькими различными именами для одного IP-адреса. Функционирование виртуальных хостов прозрачно для клиентов и воспринимается как работа с отдельными Web-сайтами при наличии одного физического сервера. Достаточно подробно настройки виртуальных хостов, как и другие вопросы конфигурирования, описаны в документации на Web-сервер Apache.

Рассмотрим некоторые практические вопросы работы с Web-сервером Apache.

Обычно первый вопрос, возникающий у пользователей, не знакомых с особенностями работы Apache, можно сформулировать так: куда поместить созданные Web-страницы и как указать URL для проверки их работоспособности. Для ответа на данный вопрос я продемонстрирую полный цикл разработки и проверки очень простой HTML-страницы.

Создадим простейшую HTTP-страницу (назовем ее test.html) и протестируем ее. Для этого при помощи текстового редактора введем следующий текст, включающий такие, например, строки:

```
<html>
<body>
<h1> TEST HTML-page OK </h1>
</body>
</html>
```

Сохраним данные в файле test.html в корневом каталоге, указанном директивой `DocumentRoot` из файла конфигурации /usr/local/apache2/conf/httpd.conf. По умолчанию `DocumentRoot` определяется как "/usr/local/apache2/htdocs".

Затем открываем окно браузера и в строке редактирования адреса набираем следующий URL:

```
http://127.0.0.1/test.html
```

Если страница не содержит ошибок, то в окне браузера будет отображена текстовая строка

```
TEST HTML-page OK
```

Во многих случаях требуется проверка работоспособности сервера Apache без использования браузера. Это может быть вызвано целым рядом причин, например, некорректной работой используемой версии браузера, необходимостью проверки специальных настроек Web-сервера, тестирования дополнительных модулей, установленных в Apache, и т. д.

Такую проверку можно выполнить при помощи программы, разработанной на языке C (назовем ее `test_apache`), которая посылает Web-серверу HTTP-запрос и ожидает от него ответа. В качестве единственного параметра программа принимает имя ресурса. Исходный текст программы, сохраненный в файле test_apache.c, приводится далее:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
```

```
#include <sys/socket.h>
#include <netdb.h>

#define port 81
char req[256];
char buf[4096];
int fdsock, ret;

int main(int argc, char* argv[])
{
 if (argc != 2)
  {
   printf("Usage: %s resource\n", argv[0]);
   exit(1);
  }
 struct sockaddr_in client;
 struct hostent *host;
 bzero(req, sizeof(req));
 bzero(&client, sizeof(client));

 int bytesRead = sprintf(req, "GET %s HTTP/1.0\n\n", argv[1]);
 printf("%s\n", req);
 client.sin_addr.s_addr = htonl(INADDR_ANY);
 client.sin_family = AF_INET;
 client.sin_port = htons(port);

 fdsock = socket(AF_INET, SOCK_STREAM, 0);
 connect(fdsock, (struct sockaddr*)&client, sizeof(client));
 send(fdsock, req, bytesRead, 0);
 do {
      ret = recv(fdsock, buf, sizeof(buf), 0);
      buf[ret] = 0;
      printf("%s\n", buf);
    } while (ret > 0);
 close(fdsock);
 return 0;
}
```

Я не буду анализировать весь исходный текст — подобные программы мы рассматривали при изучении протокола TCP, остановлюсь лишь на принци-

пах формирования HTTP-запроса в данном приложении, поскольку это самая сложная его часть. В данной программе интерпретация URL выполняется оператором

```
int bytesRead = sprintf(req, "GET %s HTTP/1.0\n\n", argv[1]);
```

Здесь в строку запроса включен параметр командной строки `argv[1]`, представляющий собой путь к требуемому ресурсу. Часть URL, представляющая собой имя хоста и порт, используется для создания сокета и подключения к Web-серверу посредством системных вызовов `socket()` и `connect()`. Обратите внимание на то, что в директиве

```
#define port 81
```

указано значение прослушивающего порта Web-сервера (в данном случае 81).

Откомпилировав программу, можно проверить ее в работе. Если Web-сервер Apache функционирует нормально, то на экране дисплея мы увидим содержимое его начальной страницы index.html, которая по умолчанию (для версий 2.2.0 и выше) находится в каталоге /usr/local/apache2/htdocs:

```
# ./test_apache /
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Thu, 04 May 2006 19:52:50 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Sat, 20 Nov 2004 20:16:24 GMT
ETag: "4a492-2c-4c23b600"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html

<html><body><h1>It works!</h1></body></html>
```

Напомню, что если расположение URL-ресурса не задано явным образом, то Web-сервер Apache выполняет его поиск в каталоге, указанном директивой `DocumentRoot` в файле конфигурации /usr/local/apache2/conf/httpd.conf.

Подобным образом можно проверить и содержимое созданной нами тестовой страницы test.html, помещенной в этот же каталог:

```
# ./test_apache /test.html
GET /test.html HTTP/1.0

HTTP/1.1 200 OK
```

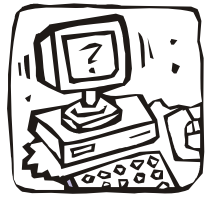
```
Date: Thu, 04 May 2006 19:55:25 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Thu, 04 May 2006 18:28:14 GMT
ETag: "4a8b5-3c-9132eb80"
Accept-Ranges: bytes
Content-Length: 60
Connection: close
Content-Type: text/html

<html>
<body>
<h1> TEST HTML-page OK </h1>
</body>
</html>
```

Поскольку многие читатели знакомы с языком Perl, я приведу исходный текст тестовой программы (она называется test_apache.pl) на этом языке:

```
#!/usr/bin/perl -w
use Socket;
my $data;
$path = shift or die "Usage: $0 path";
$req = "GET $path HTTP/1.0\n\n";
$port = 81;
$host = inet_aton("127.0.0.1");
$addr = sockaddr_in($port, $host);
socket(CLIENT, AF_INET, SOCK_STREAM, getprotobyname("tcp"))
          or die "socket: $!";
connect(CLIENT, $addr) or die "connect: $!";
print "Your request: $req";
syswrite(CLIENT, $req);
print $data while sysread(CLIENT, $data, 1024);
close(CLIENT);
```

Как и программа на C, это приложение в качестве параметра принимает имя проверяемого ресурса.

Заканчивая анализ работы с Web-сервером Apache, хочу заметить, что многие коммерческие версии серверов по своим функциональным особенностям схожи с Apache и даже заимствуют часть программных решений, разработанных для этого сервера. В этом смысле понимание особенностей функционирования Web-сервера Apache является очень полезным как для обычных пользователей, так и для разработчиков Web-сайтов и иного сетевого программного обеспечения.



Глава 11

Графические оболочки UNIX

Классическая ОС UNIX многие годы развивалась как система, в которой все операции можно было выполнять, используя текстовую консоль. Все изменилось с появлением графических интерфейсов, в особенности, графической системы X Window, на которой в той или иной степени базируются все современные графические оболочки.

Система X Window (X Window System), известная под названием X11 или X, была разработана для того, чтобы обеспечить оконный интерфейс для графических приложений, работающих в операционных системах UNIX и использующих растровые дисплеи. X обеспечивает набор стандартных библиотек и работает по стандартному протоколу, что дает возможность разрабатывать графические интерфейсы пользователя (Graphical User Interface, GUI) для широкого класса UNIX-приложений. Кроме того, X включается и в состав других UNIX-подобных операционных систем. Немаловажно и то, что все современные операционные системы UNIX поддерживают стандарты X Window.

X обеспечивает базовый набор функций для графического интерфейса пользователя: рисование и перемещение окон на экране дисплея, а также взаимодействие с клавиатурой и мышью. Отмечу очень важный момент: система X Window сама по себе не является интерфейсом пользователя — это обеспечивают клиентские программы. Сами же клиентские программы при их создании используют возможности, предоставляемые X, и могут строиться с какими угодно интерфейсами.

Особенностью X является ее прозрачность как для локальных, так и для сетевых клиентов. По своей сути X является сетевой системой, при этом термины "сервер" и "клиент" отличаются от привычных для пользователей представлений. Обычно программа-клиент выполняется на локальном компьютере и запрашивает услуги сервера, работающего на каком-нибудь удаленном компьютере сети. С системой X все наоборот — сервер находится на локальной машине, а клиент на удаленной.

Система X Window была разработана в Массачусетском технологическом институте (MIT) в 1984 году. В настоящее время система использует стандарт X11, впервые появившийся в сентябре 1987 года. Проект X Window в настоящее время сопровождает организация XOrg Foundation, выпустившая систему как свободно распространяемое программное обеспечение в соответствии с условиями лицензии MIT.

Логотипом системы X Window является пиктограмма, которую вы видите на рис. 11.1.

Рис. 11.1. Логотип X Window

Подход к разработке данной системы оправдал надежды, и эволюция пользовательских интерфейсов, построенных на основе X Window, — тому свидетельство. Начиная с версии 11, которая используется в большинстве рабочих станций UNIX, проект X Window превратился фактически в международный — в нем принимают участие университеты Карнеги-Миллан и Беркли, а также такие известные компании, как Sun Microsystems, Hewlett-Packard, Siemens и др.

В основу архитектуры системы X Window положен модульный принцип, благодаря чему достигается высокая гибкость, поскольку функции модулей (компонентов) системы четко разграничены. Одним из таких компонентов является оконный менеджер (window manager), определяющий внешний вид и интерфейсные возможности системы. Пользовательский интерфейс можно поменять простой заменой одного оконного менеджера другим.

Разработано несколько десятков различных оконных менеджеров, от самых простых (twm) до очень сложных, в которых реализованы целые интегрированные системы (KDE, GNOME и т. д.).

11.1. Модель "клиент-сервер"

Система X создавалась для работы в сети и использует модель "клиент-сервер": пользователь работает с сервером (он называется X-сервер), а программы, с которыми взаимодействует X-сервер, называются клиентами.

X-сервер является отдельным UNIX-процессом, взаимодействующим с программами-клиентами посредством пакетов данных. Если сервер и клиент находятся на разных машинах, то обмен данными осуществляется по сети, а если на одной, то используется механизм межпроцессных коммуникаций (IPC). X-сервер предоставляет свои ресурсы программам-клиентам, обеспечивающим всю остальную функциональность. Функции сервера включают управление дисплеем, обработку ввода с клавиатуры и мыши и т. д. Это означает, что существует принципиальная возможность использования ресурсов разных вычислительных машин без специальных методов при разработке программного обеспечения.

X-сервер принимает запросы от оконных интерфейсов и возвращает параметры ввода (клавиатура, мышь). Он может одновременно обслуживать многие компьютеры, обладая при этом собственной вычислительной мощностью (например, выполняться на рабочей станции). При использовании X Window сервер работает на компьютере пользователя, в то время как клиентские программы могут функционировать на других машинах. Такая конфигурация является несколько необычной, поскольку в большинстве реализаций схемы "клиент-сервер" клиент выполняется на машине пользователя, в то время как сервер работает на удаленном хосте.

В терминологии X Window удаленные программы вызывают X-сервер на локальной машине и считаются клиентами, а локальный X-дисплей принимает входящие запросы, поэтому работает как сервер. Если компьютер не подключен к сети, то сервер и клиенты X работают на нем одновременно. Важно помнить, что клиенты — это программы, создающие окна.

Схема взаимодействия клиента и сервера в системе X Window показана на рис. 11.2.

В данном примере X-сервер принимает ввод с клавиатуры и мыши и выводит информацию на дисплей. Здесь программы-клиенты (Web-браузер и эмулятор терминала) выполняются как на пользовательском компьютере, так и на удаленном. Тем не менее, все программы-клиенты, независимо от места их выполнения, управляются X-сервером.

Взаимодействие клиентов с X-сервером осуществляется посредством протокола обмена, известного под названием X-протокола. Он обеспечивает обмен данными, не зависящий от характера соединения (локальный или сетевой) — это означает, что процесс взаимодействия является "прозрачным" как для клиента, так и сервера.

Запуск программы-клиента на локальном хосте, где работает X-сервер, интуитивно понятен и не требует объяснений. Что же касается запуска удаленного клиента, то на этом следует остановиться более подробно. Запуск уда-

ленного клиента, отображающего вывод на консоль машины, где работает X-сервер, можно, например, представить последовательностью нескольких шагов:

1. Пользователь входит на машину удаленного клиента (посредством telnet или другой программы удаленного доступа).
2. Устанавливает опции вывода на машину пользователя, например, с помощью команды

   ```
   export DISPLAY=локальный_компьютер:0
   ```

   которую можно запустить из командной оболочки на удаленном клиенте.
3. Запускает клиента.

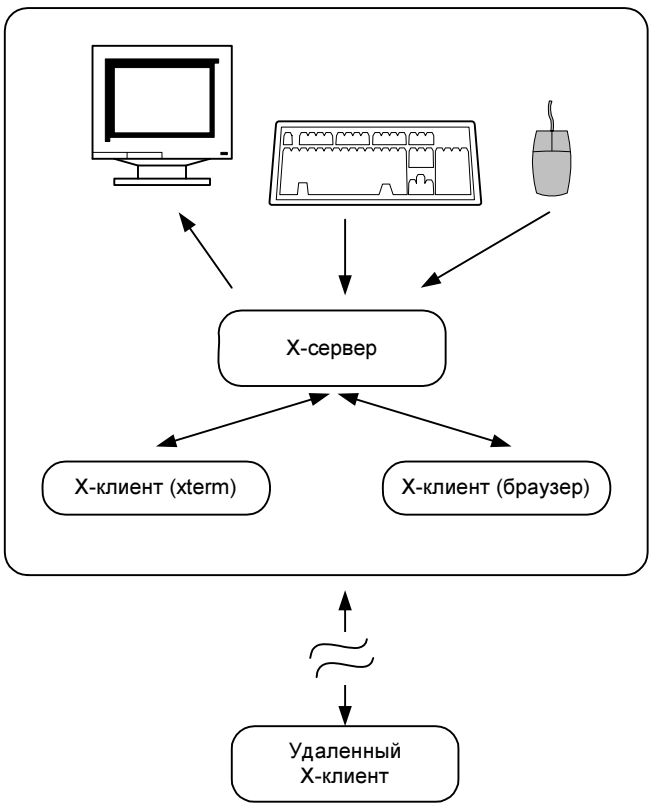


Рис. 11.2. Реализация X Window в архитектуре "клиент-сервер"

После этого удаленный клиент устанавливает соединение с локальным сервером, и удаленное приложение будет осуществлять вывод на локальный

экран и принимать ввод от устройств ввода (клавиатура, мышь). Точно так же на локальной машине можно запустить небольшую вспомогательную программу, с помощью которой можно установить соединение с удаленным клиентом и выполнить требуемую клиентскую программу.

Работа с удаленным клиентом может понадобиться при:

- ❒ администрировании удаленной машины с использованием графического интерфейса пользователя;
- ❒ выполнении программы-симулятора на удаленной UNIX-машине и отображении результатов на локальном компьютере;
- ❒ выполнении графических приложений на нескольких машинах одновременно, управляемых с одного дисплея и использующих одну клавиатуру и мышь.

Взаимодействие между X-сервером и клиентом осуществляется посредством обмена пакетами по программному каналу или по сети. Клиент устанавливает соединение с сервером, затем посылает ему первый пакет. Сервер отвечает клиенту пакетом, указывающим либо на отказ в установлении соединения, либо разрешением на создание соединения и дополнительным запросом для дальнейшей аутентификации. Если соединение разрешено, то очередные пакеты содержат данные для последующего взаимодействия с сервером.

После установки соединения между клиентом и сервером для обмена информацией используются три типа пакетов:

- ❒ запрос — клиент запрашивает информацию с сервера или требует выполнения определенного действия;
- ❒ ответ — сервер отвечает на запрос клиента; следует отметить, что не на все запросы даются ответы;
- ❒ извещение о событии — сервер информирует клиента о наступлении определенного события, например, о вводе информации с клавиатуры, о нажатии кнопки мыши или о перемещении окна и изменении его размера. Во многих случаях клиент может потребовать от X-сервера сообщить о наступлении какого-либо события другому клиенту — подобным образом осуществляется взаимодействие между клиентами. Например, когда программа-клиент запрашивает обработку выделенного фрагмента текста, информация об этом событии отправляется другой программе-клиенту, которая в данный момент управляет окном с находящимся там текстом. Информация о некоторых типах событий всегда отправляется клиенту, но в большинстве случаев такая информация отправляется только тогда, когда клиент предварительно объявил, что заинтересован в ней. Например, клиента могут интересовать только события, связанные с нажатием клавиш, и не интересовать события, связанные с мышью.

X-протокол является асинхронным — это означает, что как клиент, так и X-сервер начинают выполнять следующую операцию, не ожидая завершения предыдущей. Если соединение между клиентом и X-сервером обрывается, то X-протокол предполагает сброс установок, при этом состояние сеанса в момент потери связи не сохраняется. По этой причине применение базового варианта X Window в беспроводных системах ограничено из-за частых замираний и пропаданий сигнала.

Еще одной важной особенностью системы X Window является ее модульность, что повышает ее гибкость и возможность адаптации к разным операционным системам. Следует отметить и то, что X не интегрирована в операционную систему UNIX, а работает поверх нее, так же как и другие сервисы. Наконец, X является открытым стандартом и может функционировать на различных платформах.

Все, что вы наблюдаете на экране монитора, является результатом взаимодействия различных компонентов: операционной системы, X Window, оконного менеджера и, возможно, какой-нибудь графической оболочки наподобие GNOME или KDE. При этом отдельные компоненты (оконный менеджер, графическая оболочка и сама система X) могут настраиваться автономно от остальных, что обеспечивает высокую гибкость в настройке интерфейса пользователя, но одновременно усложняет задачу настройки всей системы.

Спецификации протокола X Window не обязывают машины клиента и сервера работать под управлением одной и той же операционной системы или даже быть одним и тем же типом компьютера. Существует возможность запускать X-сервер в Microsoft Windows или MacOS, и есть множество свободно распространяемых и коммерческих приложений, которые это реализуют.

Программирование приложений, работающих в X Window, является достаточно простым, поскольку система предоставляет для этих целей стандартную библиотеку процедур. Например, для вывода на экран точки достаточно вызвать соответствующую стандартную библиотечную процедуру, передав ей требуемые параметры: процедура выполнит всю работу по формированию пакетов данных и передаче их X-серверу.

Программный интерфейс X-сервера с соответствующим аппаратно-программным обеспечением образует в терминах X Window "дисплей", в качестве которого может выступать, например, обычный компьютер. Дисплей выполняет программный код X-сервера, обслуживая клавиатуру и позиционирующее устройство, например, мышь. Дисплей поддерживает множества физических "экранов" — аппаратно-программных комбинаций графических подсистем (видеокарта, графический акселератор) и мониторов, создающих растровое изображение. Количество обслуживаемых экранов в текущих

реализациях X Window ограничено значением 231, что является максимальным значением для 32-разрядных машин.

Система X Window предназначена для работы на растровых дисплеях, изображение в которых представляется матрицей светящихся точек — пикселов, каждый из которых кодируется определенным числом битов (как правило, 2, 4, 8, 16 или 24). Количество битов на пиксел называют толщиной или глубиной дисплея. Биты с одинаковыми номерами во всех пикселах образуют как бы плоскость, параллельную экрану, которую называют цветовой плоскостью. X позволяет рисовать в любой цветовой плоскости (или плоскостях), не затрагивая остальные.

Значение пиксела не задает непосредственно цвет точки на экране — цвет определяется с помощью специального массива данных, называемого палитрой. Цвет представлен содержимым ячейки палитры, номер которой равен значению пиксела.

X имеет большой набор процедур, позволяющих рисовать графические примитивы — точки, линии, дуги, выводить текст и работать с областями произвольной формы.

Каждый экран дисплея может содержать множество перекрывающихся окон — базовых элементов системы, представляющих собой прямоугольные или произвольной формы (с учетом расширений системы X Window) отображаемые области растровой памяти. Система X Window использует окно для вывода графической информации, предоставляя его программе-клиенту.

Любое созданное окно должно иметь родительское окно — тем самым создается иерархия или дерево окон. В вершине этой иерархии находится так называемое корневое окно, которое автоматически создается X-сервером. Визуально корневое окно занимает всю площадь экрана и находится позади всех остальных окон, а сами окна могут располагаться на экране произвольным образом, перекрывая друг друга.

Иерархию окон демонстрирует рис. 11.3.

На этой схеме показан один из примеров расположения окон. Здесь (1) — корневое окно, занимающее экран целиком, (2) и (3) — окна переднего плана, а (4) и (5) являются подокнами окна (2). При этом части некоторых окон выходят за пределы родительского окна, поэтому их не видно (изображены пунктиром).

С каждым окном связана система координат, начало которой находится в левом верхнем углу окна — при этом ось x направлена вправо, ось y вниз, а единицей измерения по обеим осям является пиксел. Окно имеет внутреннюю область и рамку, а его основными атрибутами являются ширина и высота

внутренней области, а также ширина (толщина) рамки. Эти параметры называются геометрией окна.

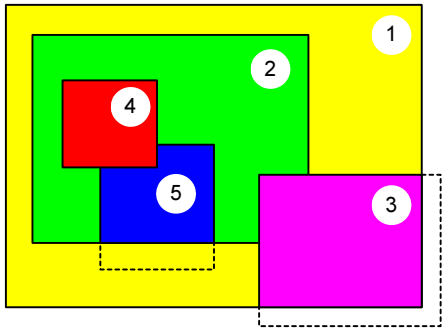


Рис. 11.3. Пример иерархии окон в X Window

Следует отметить, что графические элементы управления, такие, например, как кнопки, меню, значки и другие, также реализованы с использованием окон.

Здесь хотелось бы отметить одну важную особенность X Window: если не используются дополнительные программы-клиенты (оконные менеджеры или графические оболочки), то X-сервер при запуске создает корневое окно без рамок, значков и всплывающих меню. Пример корневого окна при запуске Cygwin (эмулятор UNIX) представлен на рис. 11.4.

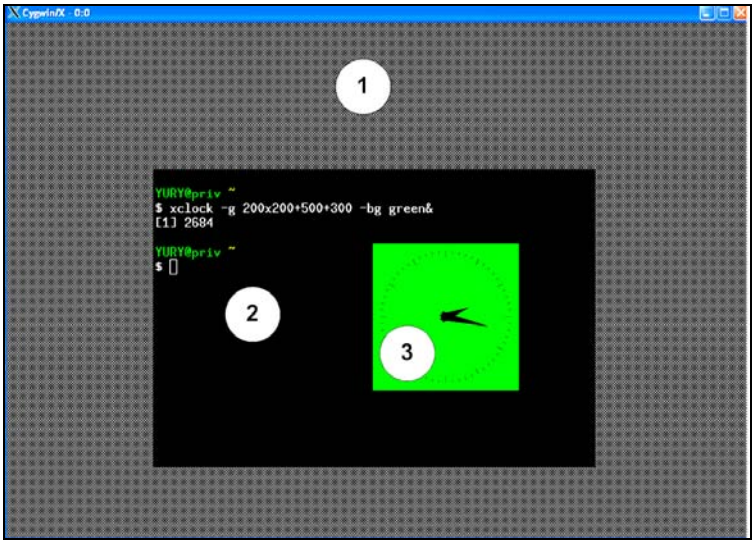


Рис. 11.4. Вид корневого окна, созданного X-сервером

Здесь для создания корневого окна (1) была использована команда

```
# xinit -g 60x20+200+200 -fn 10x20 -bg black -fg white -display :0
```

При этом был запущен X-сервер на дисплее 0, и, поскольку `xinit` автоматически вызвала программу эмулятора терминала `xterm` с указанной геометрией и параметрами шрифта и цветов, было создано окно (2) приложения `xterm`. После этого из окна эмулятора `xterm` было запущено приложение `xclock` (3).

Программный интерфейс X Window позволяет программе-клиенту выполнять целый ряд операций как с окнами, так и с их содержимым — окна можно создавать, перерисовывать на экране, удалять, в них можно выводить текст и графические примитивы. Особенностью системы является то, что в ней не предусмотрен встроенный механизм для управления окнами с помощью мыши или клавиатуры — эти функции выполняет оконный менеджер.

Взаимодействие X и клиентских программ, включая оконные менеджеры, будет рассмотрено далее в этой главе.

11.2. Запуск и настройка X Window

Существуют два основных метода установки соединения с X-сервером и инициализации начальных установок для клиентских программ. Выбор того или иного варианта определяется тем, какая операционная система запущена на компьютере, и используются ли другие графические системы кроме X Window.

Если необходимо, чтобы система X Window работала постоянно в данной операционной системе, можно настроить машину так, чтобы при начальной загрузке системы запускался менеджер дисплея `xdm`, который берет на себя ответственность за бесперебойное функционирование X-сервера, а также позволяет пользователю войти в систему. Если `xdm` запущен, то вы увидите на экране приглашение ко вводу имени и пароля.

Для входа в систему следует, как обычно, ввести имя пользователя и пароль. После успешного входа в систему `xdm` создает окружение для работы графической подсистемы. Если в домашнем каталоге пользователя есть файл .xsession, то менеджер `xdm` интерпретирует данный файл как командный и использует его для запуска X-клиентов (эмуляторов терминала, часов, оконного менеджера, пользовательских настроек, например, фона рабочего стола и т. д.).

Второй вариант запуска графической системы применяется в том случае, когда в операционной системе имеется несколько оконных систем, и предпола-

гает запуск X Window через программу `xinit`, если таковая имеется в системе. Эту программу можно запустить из командного интерпретатора. `xinit`, в свою очередь, запускает дополнительные программы, которые выполняют специфичную для хоста инициализацию (загрузку необходимых ресурсов, запуск оконного менеджера, отображение часов, запуск эмуляторов терминала оптимальным образом).

Описание настроек и начального набора X-клиентов содержится в соответствующих файлах конфигурации. При запуске система X Window считывает файлы конфигурации (как пользовательские, так и общесистемные) и запускает указанные пользователем X-клиенты. Даже если X Window не запускается автоматически при входе в систему, можно запустить ее из командной строки в окне терминала, работающего в текстовом режиме, выполнив команду `startx`. Если появляется сообщение, что такой команды нет ("command not found"), нужно воспользоваться командой `xinit`.

Если же ни одна из этих команд не работает, возможно, в операционной системе не установлена X Window. В этом случае придется подробно изучить документацию на используемый дистрибутив операционной системы.

Обобщая, можно сказать, что сессия работы пользователя в системе X Window начинается после запуска X-сервера или ввода пользовательского идентификатора и пароля в специальную форму менеджера экрана системы X Window (`xdm`, `kdm`, `gdm` или другой) и заканчивается после завершения работы X-сервера. Сеанс работы в ОС UNIX при этом может закончиться (при наличии менеджера экрана) или продолжиться.

Завершить работу в X Window можно несколькими способами. Можно закончить сеанс, нажав кнопку завершения на панели инструментов виртуального рабочего стола. Быстрый выход из X Window — нажатие комбинации клавиш <Ctrl>+<Alt>+<Backspace>.

Далее рассмотрим более детально некоторые ключевые программы, используемые при запуске системы X Window.

11.2.1. Команда *startx*

Вначале рассмотрим один из распространенных способов запуска X-сервера с помощью команды `startx`, имеющей следующий синтаксис:

```
startx [ [client] options ... ] [ -- [server] options ... ]
```

Команда `startx`, помимо других операций по инициализации, запускает на исполнение двоичный файл `xinit`. Основное назначение последнего — запуск X-сервера. Поскольку команда `startx`, кроме вызова `xinit`, выполняет и другие необходимые действия, то запуск системы X Window с помощью

этой команды является предпочтительным. Если данная команда не запускается, то это означает, что система X Window не установлена, или возникли технические неполадки, препятствующие запуску X-сервера. Еще одной причиной может быть ошибка в конфигурации (например, установлены такие режимы работы, которые не соответствуют возможностям графической подсистемы).

Если дополнительные программы отсутствуют, то вы можете создать командный файл, использующий утилиту `xinit`, самостоятельно. Естественно, что разработать командный файл для запуска X-сервера и всех клиентов может только опытный пользователь.

В зависимости от конфигурации системы X Window может стартовать сразу же после загрузки операционной системы или запускаться самим пользователем.

Для запуска X Window необходимо несколько файлов конфигурации. Эти файлы используются командами `startx` и `xinit`, а наиболее важным из них является файл .xinitrc.

Это командный файл, передаваемый `xinit` посредством команды `startx`. Он содержит настройки некоторых общих (глобальных) ресурсов, таких как хранитель экрана, клавиатура и др., и является основным для всего процесса запуска.

Еще одним файлом, который необходим при запуске X, является Xclients. Он находится в домашнем каталоге пользователя и используется для запуска клиентов, специфичных для данного пользователя, в отличие от .xinitrc, содержащего общие настройки.

Файл Xresources содержит программные настройки, позволяющие изменить заданные по умолчанию значения. Он обычно применяется для определения размеров шрифтов, цветов и общего внешнего вида. Кроме этого, его можно использовать для получения графических эффектов.

Рассмотрим более подробно процесс запуска. После выполнения команды `startx` команда `xinit`, выполняющая основную часть процесса запуска X, просматривает файл /etc/X11/xinit/.xinitrc (глобальный) либо файл ~/.xinitrc (локальный), если он существует. Если ни один из файлов не найден, то выбор настроек выполняет сам `xinit`, хотя это и не лучший вариант. Командный файл .xinitrc запускает X-клиентов и в завершение процесса — указанный в последней строке исполняемый файл оконного менеджера. Следует заметить, что приоритет локального файла .xinitrc выше, чем у глобального. Для запуска оконного менеджера обычно применяется команда `exec`.

Вот пример содержимого простейшего файла .xinitrc:

```
# /etc/X11/xinit/.xinitrc
# !/bin/sh
exec /usr/X11R6/bin/fvwm2
```

Здесь выполняется всего одно действие — запускается оконный менеджер `fvwm2` (Feeble Virtual Window Manager 2). Можно настроить интерфейс X Window для себя, установив, например, определенные параметры экрана и клавиатуры. Эти настройки будут выполняться автоматически каждый раз при запуске X Window.

Вот пример такого командного файла:

```
#/etc/X11/xinit/.xinitrc
#!/bin/sh

# Выполним установку клавиши <BackSpace>
xmodmap -e "keycode 22=BackSpace"

# Установим фон
xsetroot -solid LightSlateGrey

# Хранитель экрана должен отработать через 10 минут
xset s 600

# Запуск X-клиентов
xterm -g 80x20+150+8 &       # запуск окна терминала
xterm -g 80x20+150+325 &     # запуск второго окна под первым
xload -g +4+0 &              # индикатор загрузки процессора
xclock -g +815+0 -digital &  # запуск цифровых часов

# Запуск оконного менеджера
exec fvwm2
```

С помощью этого файла устраняется проблема замены функции клавиши <Backspace> (в некоторых системах ее функции отличаются от стандартных), удаляются муаровые "обои" и запускаются две терминальные программы.

Следует отметить, что локальный файл .xinitrc из домашнего каталога пользователя будет использоваться всегда, при этом общесистемные установки не берутся во внимание. Ключ `-g` позволяет определять геометрию для многих программ, работающих в X. Формат данного ключа следующий:

```
Length x Width + X_coordinate + Y_coordinate
```

Нужно быть внимательным при использовании команды `xterm`. Для некоторых терминалов значение ширины и высоты указывается в символах, в то время как большинство программ использует для этой цели пикселы, из-за чего окна могут получиться очень маленькими.

При редактировании командного файла каждую команду, кроме запускающей оконный менеджер, нужно завершать символом амперсанда `&`. В этом случае все программы будут запущены в фоновом режиме, что гарантирует выполнение последующих команд. Если этого не сделать, следующая команда может не выполниться вообще. Это не относится к оконному менеджеру, который должен запускаться в основном режиме, что позволяет завершить все процессы, запущенные через .xinitrc , при выходе из X.

Еще одним важным файлом для функционирования клиентов является .Xresources. Ресурсы в X Window определяют программные настройки локально (~/.Xresources) или глобально (/usr/X11R6/lib/X11/xinit/.Xresources). Графические программы включают в себя так называемые виджеты (от англ. *widgets* — элементы, объекты). Виджеты — это элементы управления (в Microsoft Windows они имеют название "controls"), такие как кнопки, полосы прокрутки, радиокнопки, окна редактирования, пиктограммы и т. д.

Виджеты обычно имеют иерархическую структуру — внутри виджета для меню может, например, находиться несколько виджетов для отдельных кнопок. Чтобы полностью определить изменения для конкретного виджета, нужно указать полный путь к индивидуальному компоненту.

Наиболее популярным в настоящее время является набор виджетов Qt. Он используется большинством оконных менеджеров, а также графическими оболочками GNOME и KDE. В случае успешного старта системы X Window происходит последовательное выполнение действий, необходимых для инициализации графического режима и запуска начального набора X-клиентов. Это занимает определенное время, тем меньшее, чем мощнее аппаратные ресурсы компьютера.

11.2.2. Программа *xinit*

Программа `xinit`, так же как и `startx`, используется для запуска сервера X Window. Кроме того, `xinit` является первой клиентской программой в системах, которые не могут запускать X-сервер непосредственно через /etc/init или в операционных средах с возможностью запуска разных оконных систем. Программа имеет следующий синтаксис:

```
xinit [ [ client ] options ] [ -- [ server ] [ display ] options ]
```

Если в командной строке не задано имя `client` клиентской программы, `xinit` пытается найти в домашнем каталоге пользователя и запустить командный файл .xinitrc, который содержит клиентские программы. Если такой файл отсутствует, то `xinit` по умолчанию выполнит команду

```
xterm -geometry +1+1 -n login -display :0
```

Если же в командной строке не задан сервер `server`, `xinit` пытается найти в домашнем каталоге пользователя и выполнить командный файл .xserverrc, чтобы запустить сервер. Если данный файл отсутствует, то по умолчанию `xinit` выполнит команду

```
X :0
```

Обратите внимание на то, что программа называется X, хотя обычным названием является `Xdisplaytype`, где `displaytype` представляет собой тип графического дисплея, который управляется данным сервером.

Хочу также обратить внимание на то, что все программы, кроме последней, прописанные в командном файле .xinitrc, должны запускаться в фоновом режиме — если этого не сделать для какой-то программы, то следующая за ней не сможет выполниться никогда. Последняя программа, включенная в командный файл, является или эмулятором терминала, или оконным менеджером и должна запускаться как обычная программа — в этом случае командный файл не закончит работу. Это означает, в свою очередь, что пользователь будет продолжать работать в графическом режиме, пока не захочет выйти из него.

Далее приводятся примеры использования программы `xinit` с аргументами командной строки.

Пример 1.

```
# xinit
```

Данная команда позволяет запустить X-сервер и выполнить пользовательский командный файл ./xinitrc (если он существует), в противном случае запускается эмулятор терминала `xterm`.

Пример 2.

```
# xinit -- /usr/X11R6/bin/Xqdss :1
```

Здесь показан запуск специфичного типа сервера на альтернативном дисплее.

Пример 3.

```
# xinit -geometry =80x65+10+10 -fn 8x13 -j -fg white -bg navy
```

Здесь запускается сервер X, после чего выполняется команда `xterm` с указанными в командной строке параметрами. Командный файл .xinitrc в этом случае игнорируется.

Пример 4.

```
# xinit -e widgets -- ./Xsun -l -c
```

В данном примере командой `./Xsun -l -c` запускается сервер и, кроме того, эмулятору терминала `xterm` передаются параметры `-e widgets`.

Пример 5.

```
# xinit /usr/ucb/rsh fasthost cpupig -display ws:1 -- :1 -a 2 -t 5
```

Здесь на дисплее 1 стартует сервер X с аргументами `-a 2 -t 5`, а также запускается команда `cpupig` на удаленной машине `fasthost`.

Пример 6.

Далее показан пример файла .xinitrc, с помощью которого запускаются часы и несколько терминалов, после чего выполняется программа оконного менеджера, являющаяся последней в командном файле. Вспомним, что последняя команда этого командного файла должна работать, в отличие от остальных, в обычном режиме.

В случае если оконный менеджер сконфигурирован правильно, выйти из графического режима можно по команде `exit`, которая и завершает работу X-сервера.

```
xrdb -load $HOME/.Xresources
xsetroot -solid gray &
xclock -g 50x50-0+0 -bw 0 &
xload -g 50x50-50+0 -bw 0 &
xterm -g 80x24+0+0 &
xterm -g 80x24+0-0 &
twm
```

11.2.3. Дополнительные настройки X-сервера

Рассмотрим некоторые параметры, используемые для запуска X-сервера. Каждый X-сервер имеет имя дисплея, представленное следующим образом:

```
имя_хоста:номер_дисплея:номер_экрана
```

Эта информация нужна приложению для того, чтобы определить, как соединиться с сервером и какой экран использовать по умолчанию (для дисплеев

с несколькими мониторами). Здесь параметр имя_хоста указывает на имя машины, к которой физически подключен дисплей. Если данный параметр не указан, то предполагается, что используется локальный X-сервер, работающий на данном компьютере.

Параметр номер_дисплея применяется для ссылки на группу мониторов, которые используют общую клавиатуру и мышь. В подавляющем большинстве случаев компьютер имеет только один дисплей, но некоторые многопользовательские системы работают с несколькими дисплеями, поэтому для того, чтобы различать их, требуется указывать так называемый номер дисплея. Нумерация дисплеев начинается с нуля. Номер дисплея является обязательным параметром и должен задаваться всегда.

Параметр номер_экрана необходим в тех случаях, когда одна клавиатура и мышь используются для двух и более мониторов. Поскольку каждый монитор работает с собственным набором окон, то каждому экрану присваивается номер экрана в момент запуска X-сервера. Если номер экрана не задан, то будет использоваться экран с номером 0.

Для POSIX-совместимых систем имя дисплея, используемое по умолчанию, задается в переменной окружения DISPLAY. Значение данной переменной устанавливается автоматически утилитой xterm эмулятора терминала, хотя иногда может потребоваться установить значение этой переменной вручную, например, при регистрации на удаленном компьютере. Сделать это можно, например, при помощи команд

```
setenv DISPLAY myworkstation:0
DISPLAY=myworkstation:0; export DISPLAY
```

Здесь переменной DISPLAY присваивается значение myworkstation:0, где myworkstation — имя локального компьютера, на котором работает X-сервер.

X-серверы в процессе взаимодействия с клиентами могут использовать самые разнообразные механизмы межпроцессных коммуникаций (потоки, разделяемую память и т. д.). По этой причине переменная имя_хоста служит для указания типа коммуникационного канала (или, как его еще называют, транспортного уровня). Поддерживаются следующие типы коммуникационных каналов:

- local — указывает на имя хоста (локальный транспорт), которое может быть и пустой строкой, например, :0, :1, и :0.1;
- tcpip — здесь в качестве имени_хоста в полном имени дисплея указывается IP-адрес машины, где размещается X-сервер. В качестве имени_хоста может использоваться также и полное интернет-имя (FQDN, Fully

Qualified Domain Name) или аббревиатура, как это показано в следующем примере:

```
x.org:0, expo:0, 198.112.45.11:0, bigmachine:1, and hydra:0.1.
```

X-сервер позволяет установить несколько уровней безопасности (access control), наиболее важными из которых являются:

- упрощенный уровень прав доступа;
- уровень, базирующийся на протоколе DES (XDM-AUTHORIZATION-1);
- уровень, базирующийся на разработке фирмы Sun Microsystems (SUN-DES-1);
- уровень, базирующийся на протоколе Kerberos (MIT-KERBEROS-5 Kerberos Version 5).

Управление правами доступа к X-серверу осуществляет программа `xdm`, размещая информацию для авторизации пользователя в доступном для него файле. Обычно список хостов, которым разрешается доступ к серверу, пуст, поэтому пользователи должны явным образом устанавливать соединение.

Для контроля доступа программа `xdm` использует файл $HOME/.Xauthority, обновляя записи авторизации в момент входа пользователя на сервер.

Если с сервером работает несколько хостов, которые разделяют общий домашний каталог (например, посредством NFS), то нет необходимости в файлах авторизации — X Window самостоятельно выполняет все требуемые манипуляции. Пользователю, кроме того, предоставляется возможность самостоятельно работать с файлами авторизации посредством утилиты `xauth`.

С помощью этой программы пользователь может извлекать записи из одного файла авторизации и включать их в другие файлы. Например, при входе на удаленные машины, которые не используют общий домашний каталог на вашем локальном хосте, вы можете отправлять им информацию об авторизации. При этом нужно учитывать, что такая информация не шифруется и передается по сети в открытой форме. Более подробная информация об установке прав доступа к X-серверу находится в man-страницах (см. Xsecurity).

Одним из существенных преимуществ системы X Window является то, что она никаким образом не ограничивает приложения ни в плане размещения окон, ни их размерами на экране, что не всегда получается, если использовать аппаратные терминальные соединения. Несмотря на то, что расположением окон на дисплее управляет оконный менеджер, большинство приложений, работающих в X Window, могут устанавливать параметры окон из командной строки в формате

```
--geometry WIDTH x HEIGHT + XOF + YOF
```

Параметры WIDTH, HEIGHT, XOF и YOF задаются в виде чисел и определяют размер и размещение главного окна приложения.

Параметры WIDTH и HEIGHT обычно определяются или в пикселах, или в символах, в зависимости от особенностей работы приложения. Параметры XOF и YOF измеряются в пикселах и используются для позиционирования окна от левой/правой или верхней/нижней границ экрана соответственно.

Смещение XOF можно задавать в следующих формах:

- +XOF — левая кромка окна размещается на расстоянии XOF пикселов от левого края экрана. Иными словами, это значение представляет собой координату *x* данного окна. Следует сказать, что XOF может быть отрицательным числом — в этом случае левая кромка окна будет находиться вне экрана;
- -XOF — правая кромка окна размещается на расстоянии XOF пикселов от правого края экрана. XOF может быть отрицательным числом — в этом случае правая кромка окна будет находиться вне экрана.

Смещение по вертикали определяется следующими значениями:

- +YOF — верхняя кромка окна размещается на расстоянии YOF пикселов ниже от верхнего края экрана, причем данное значение представляет собой координату y окна. YOF может быть отрицательным числом — в этом случае верхняя кромка окна будет находиться вне экрана;
- -YOF — нижняя кромка окна размещается на расстоянии YOF пикселов выше от нижнего края экрана. YOF может быть отрицательным числом — в этом случае нижняя кромка окна будет находиться вне экрана.

Смещения могут быть представлены в виде пары значений, например:

- +0+0 — верхний левый угол;
- -0+0 — правый верхний угол;
- -0-0 — правый нижний угол;
- +0-0 — левый нижний угол;

В следующем примере окно эмулятора терминала размещается приблизительно в центре экрана, а окна монитора производительности, почты и часов находятся в правом верхнем углу экрана:

```
xterm -fn 6x10 -geometry 80x24+30+200 &
xclock -geometry 48x48-0+0 &
xload -geometry 48x48-96+0 &
xbif -geometry 48x48-48+0 &
```

11.3. Команды X Window и настройки параметров системы

В системе X Window имеется множество команд, позволяющих настроить графический интерфейс пользователя. Как и любые другие команды операционной системы UNIX, они могут быть использованы в командных сценариях.

Необходимо учитывать, что эти команды работают в любых графических оболочках и с любыми оконными менеджерами, поскольку являются встроенными в X Window. Вне зависимости от того, какой оконный менеджер или графическая оболочка применяется, команды X Window можно использовать для настройки и оптимизации как пользовательского интерфейса, так и графической системы в целом! Вот перечень наиболее важных команд, разработанных X Консорциумом:

- `xterm` — эмулятор терминала;
- `xdm` — менеджер дисплея;
- `xconsole` — команда перенаправления вывода консоли;
- `xmh` — интерфейс почтовых служб;
- `bitmap` — редактор ресурсов;
- `edires` — редактор ресурсов;
- `xauth`, `xhost` — команды управления доступом к программам;
- `xset` — позволяет произвести пользовательские настройки для работы с X Window;
- `xrdb`, `xcmsdb`, `xset`, `xsetrot`, `xstdcmap` и `xmodmap` — программы для настройки интерфейса пользователя.

Кроме того, имеется целый ряд очень полезных утилит пользователя, позволяющих выполнить настройки графического интерфейса и устройств (клавиатуры, мыши, экрана). Вот наиболее важные из них и их функции:

- `xmodmap` — выполняет настройку таблицы кодировки и клавиатуры, а также управляет переназначением клавиш;
- `xfd`, `xlsfonts` — отображают информацию о шрифтах, доступных пользователю;
- `xwd`, `xwud` — утилиты управления экраном;
- `xmag` — монитор производительности;
- `xfs` — редактор шрифтов и другие вспомогательные программы: `fsinfo`, `fslsfonts`, `fstobdf`;

- `xkbcomp` — редактор таблиц раскладки клавиатуры. Сюда же можно отнести и другие утилиты работы с клавиатурой: `xkbcomp`, `xkbprint`, `xkbbell`, `xkbevd`, `xkbvleds` и `xkbwatch`;
- `xkill` — утилита завершения сеанса работы пользователя;
- `rstart`, `xon` — программы удаленного доступа и выполнения программ;
- `lbxproxy` — выполняет оптимизацию настройки прокси для X-протокола;
- `proxymngr` — контроль и управление настройками X-прокси;
- `xfwp` — брандмауэр;
- `twm` — оконный менеджер;
- `clocks`, `xclock` и `oclock` — утилиты настройки даты/времени;
- `xfd` — выводит информацию об используемых шрифтах;
- `xlsfonts`, `xwininfo`, `xlsclients`, `xdpyinfo`, `xlsatoms`, `xprop` — утилиты, выводящие информацию о шрифтах, окнах и дисплеях;
- `xwd`, `xwud` и `xmag` — утилиты настройки изображения (screen image manipulation utilities);
- `x11perf` — утилита для измерения производительности;
- `bdftopcf` — компилятор шрифтов;
- `Xserver`, `rgb`, `mkfontdir` — сервер дисплея и вспомогательные утилиты;
- `rstart` и `xon` — утилиты для запуска удаленных программ;
- `xclipboard` — менеджер буфера обмена (clipboard manager);
- `xkbcomp`, `xkbprint`, `xkbel`, `xkbevd`, `xkbvleds` и `xkbwatch` — компилятор раскладки клавиатуры (keyboard description compiler) и вспомогательные утилиты.

Существует и масса других утилит, оконных менеджеров и других вспомогательных программ, которые можно загрузить из Интернета как свободно распространяемое программное обеспечение.

Команды X Window можно выполнять как из командной строки, так и в командных сценариях. Вот несколько примеров командных строк с утилитами X Window:

```
$ xrdb $HOME/.Xresources
$ xmodmap -e "keysym BackSpace = Delete"
$ mkfontdir /usr/local/lib/X11/otherfonts
$ xset fp+ /usr/local/lib/X11/otherfonts
$ xmodmap $HOME/.keymap.km
```

```
$  xsetroot -solid 'rgbi:.8/.8/.8'
$  xset b 100 400 c 50 s 1800 r on
$  xset q
$  twm
$  xmag
$  xclock -geometry 48x48-0+0 -bg blue -fg white
$  xeyes -geometry 48x48-48+0
$  xbiff -update 20
$  xlsfonts '*helvetica*'
$  xwininfo -root
$  xdpyinfo -display joesworkstation:0
$  xhost -joesworkstation
$  xrefresh
$  xwd | xwud
$  bitmap companylogo.bm 32x32
$  xcalc -bg blue -fg magenta
$  xterm -geometry 80x66-0-0 -name myxterm $*
$  xon filesysmachine xload
```

Рассмотрим практические примеры настройки отдельных параметров X Window при помощи некоторых из перечисленных выше утилит. Читатели смогут более детально ознакомиться со всеми возможностями команд X Window, обратившись к соответствующим man-страницам и другой документации.

11.3.1. Команда *xset*

Команда `xset` предназначена для настройки параметров дисплея и имеет синтаксис:

```
xset [-display дисплей] [-b] [b on|off] [b [volume [pitch [duration]]]
     [[-]bc] [-c] [c on|off] [c [volume]] [[+-]dpms]
     [dpms standby [suspend [off]]]
     [dpms force standby|suspend|off|on]
     [[-+]fp[-+=] путь[,путь[,...]]] [fp default] [fp rehash]
     [[-]led [integer]]  [led on|off]  [m[ouse]  [accel_mult[|accel_div]
     [threshold]]]  [m[ouse] default] [p цвет_пиксела]
     [[-]r [код_клавиши]]  [r on|off]  [r rate delay [rate]]
     [s [время [периодичность]]]
     [s blank|noblank] [s expose|noexpose] [s on|off] [s default]
     [s activate] [s reset] [q]
```

Вот смысл некоторых опций:

- `-display` *дисплей* — определяет используемый X-сервер;
- `b` — устанавливает параметры для динамика (звук, тон, частоту);
- `-dpms` — запрещает использование DPMS (Energy Star);
- `+dpms` — разрешает использование DPMS;
- `dpms` *флаги* — устанавливает параметры для режима DPMS. Опция может принимать до трех числовых параметров. Например, параметр `force` требует немедленного переключения в требуемое или установленное состояние. Для режима возможно одно из состояний: `standby`, `suspend`, `off` или `on`;
- `fp=`*путь*`,...` — определяет путь к файлам шрифтов. Этот параметр интерпретируется только сервером, но не клиентом. Обычно путь представляет собой имя каталога;
- `fp default` — восстанавливает умолчания для пути поиска файлов шрифтов;
- `fp rehash` — требует от сервера реинициализации базы данных шрифтов. Такая опция задается в том случае, если добавлен новый шрифт;
- `-fp` — удаляет указанные путевые имена к файлам шрифтов;
- `fp+` — добавляет путевые имена к файлам шрифтов;
- `led` — управляет светодиодами (LEDs) на клавиатуре. С помощью этой опции можно включать или отключать индикацию клавиатуры;
- `m` — устанавливает параметры мыши. Используется для настройки "ускорения" и "чувствительности" мыши. Обычно установкой этих параметров пытаются добиться более точного позиционирования указателя мыши;
- `r` — устанавливает опции автоповторения символов клавиатуры. Установка в `-r` или `r off` запрещает автоповторение, в то время как установка `r` или `r on` разрешает автоповторение. Если после опции `-r` или `r` следует число в диапазоне от 0 до 255, то это запрещает или разрешает автоповторение для клавиши с соответствующим кодом. Коды клавиш со значением меньше 8 обычно не используются. Например, команда

  ```
  xset -r 10
  ```

  запрещает автоповторение для клавиши <1> на обычной IBM-клавиатуре;
- `s` — устанавливает параметры для хранителя экрана (screen saver). Опция может принимать до двух числовых параметров. Этими параметрами могут быть: `blank` или `noblank`, `expose` или `noexpose`, `on` или `off`, `activate`, `reset` или `default`. Если параметры не заданы, по умолчанию принимается `default`;
- `q` — выводит на экран информацию о текущих настройках.

Следует заметить, что все настройки этой команды устанавливаются в значения по умолчанию после завершения сеанса работы в X Window. Необходимо также учитывать, что не все реализации X Window поддерживают те или иные опции.

Рассмотрим несколько примеров применения команды `xset`. Следующая команда выводит информацию о текущих настройках сеанса X Window:

```
# xset q
Keyboard Control:
  auto repeat:  on    key click percent:  0    LED mask:  00000000
  auto repeat delay:  500    repeat rate:  30
  auto repeating keys:  00ffffffdffffbbf
                        fadfffffffdfe5ff
                        fffffffffffffff
                        fffffffffffffff
  bell percent:  50    bell pitch:  400    bell duration:  100
Pointer Control:
  acceleration:  2/1    threshold:  4
Screen Saver:
  prefer blanking:  yes    allow exposures:  yes
  timeout:  600    cycle:  600
Colors:
  default colormap:  0x20    BlackPixel:  0    WhitePixel:  16777215
Font Path:
  /root/.gnome2/share/cursor-fonts,unix/:7100,/root/.gnome2/share/fonts
Bug Mode: compatibility mode is disabled
DPMS (Energy Star):
  Standby: 1200    Suspend: 1800    Off: 2400
  DPMS is Enabled
  Monitor is On
Font cache:
  hi-mark (KB): 5120  low-mark (KB): 3840  balance (%): 70
File paths:
  Config file:  /etc/X11/XF86Config
  Modules path: /usr/X11R6/lib/modules
  Log file:     /var/log/XFree86.0.log
```

В следующем примере хранитель экрана будет активизироваться через 10 минут:

```
# xset s 600
```

11.3.2. Команда *xmodmap*

Одной из очень полезных команд системы X Window является команда `xmodmap`. С ее помощью можно выполнить модификацию раскладки клавиатуры и переназначение клавиш. Команду `xmodmap`, так же как и `xset`, можно использовать для адаптации интерфейса пользователя. Эта команда обычно размещается в стартовом командном файле пользователя и запускается каждый раз, когда выполняется вход в систему. Команда имеет следующий синтаксис:

```
xmodmap [-опции ...] [имя_файла]
```

Рассмотрим назначение некоторых опций этой команды:

- `-display` *дисплей* — определяет хост и дисплей, для которых выполняются установки;
- `help` — выводит краткое описание команды и ее опций;
- `-e` *выражение* — определяет выражение, которое должно быть выполнено. Само выражение может задаваться в командной строке;
- `-pk` — указывает на необходимость вывода таблицы текущей раскладки клавиатуры на устройство стандартного вывода.

В команде `xmodmap` могут использоваться выражения, с помощью которых можно переназначать функции отдельных клавиш или присваивать функции одних клавиш другим. Например, такое выражение, как

```
keycode NUMBER = KEYSYMNAME ...
```

определяет перечень символических кодов клавиш (`keysyms`), которые могут быть присвоены указанному коду клавиши (`keycode`).

Рассмотрим несколько примеров использования команды `xmodmap`. Команда `xmodmap`, заданная без параметров, направляет на стандартное устройство вывода раскладку управляющих клавиш для данного сеанса X Window:

```
# xmodmap
xmodmap:  up to 2 keys per modifier, (keycodes in parentheses):

shift       Shift_L (0x32),  Shift_R (0x3e)
lock        Caps_Lock (0x42)
control     Control_L (0x25),  Control_R (0x6d)
mod1        Alt_L (0x40),  Alt_R (0x71)
mod2        Num_Lock (0x4d)
mod3
mod4        Super_L (0x73),  Super_R (0x74)
mod5
```

В следующем примере с помощью команды

```
xmodmap -pk
```

выводится таблица с раскладкой клавиатуры, в которой указаны символические имена, присвоенные клавишам. Далее приводится часть таблицы:

```
# xmodmap -pk
There are 4 KeySyms per KeyCode; KeyCodes range from 8 to 255.

    KeyCode     Keysym (Keysym) ...
    Value       Value   (Name)  ...

      8
      9         0xff1b (Escape)
     10         0x0031 (1)      0x0021 (exclam)
     11         0x0032 (2)      0x0040 (at)
     12         0x0033 (3)      0x0023 (numbersign)
     13         0x0034 (4)      0x0024 (dollar)
     14         0x0035 (5)      0x0025 (percent)
     15         0x0036 (6)      0x005e (asciicircum)
     16         0x0037 (7)      0x0026 (ampersand)
     17         0x0038 (8)      0x002a (asterisk)
     18         0x0039 (9)      0x0028 (parenleft)
     19         0x0030 (0)      0x0029 (parenright)
     20         0x002d (minus)  0x005f (underscore)
     21         0x003d (equal)  0x002b (plus)
     22         0xff08 (BackSpace)      0xfed5 (Terminate_Server)
     23         0xff09 (Tab)    0xfe20 (ISO_Left_Tab)
     24         0x0071 (q)      0x0051 (Q)
     25         0x0077 (w)      0x0057 (W)
```

Команда `xmodmap` может применяться в командных файлах при настройке рабочего окружения пользователя, а также для разработки служебных утилит. Например, если требуется определить ASCII-код некоторой клавиши, можно выполнить командную строку

```
# xmodmap -pk|grep '(a)'|cut -f2|cut -c1-6
0x0061
```

В этом примере на экран выводится код клавиши <A>.

Простая программа (назовем ее `xmodmap_demo`) помогает определить код клавиши, заданной как параметр в командной строке:

```
if [ $# -ne 1 ]
```

```
 then
  echo "Usage: $0 [char]"
else
 x=`xmodmap -pk|grep '('$1')'|cut -f2-3`
 echo $x
fi
```

Результат работы командного файла для клавиши <k> показан далее:

```
# ./xmodmap_demo k
0x006b (k) 0x004b (K)
```

Часто бывает очень удобно функции одной клавиши передать другой. Команда `xmodmap` позволяет это сделать довольно просто:

```
# xmodmap -e "keysym BackSpace = Delete"
```

После выполнения этих команд клавиша <Backspace> будет работать как <Delete>, т. е. удалять символы, расположенные за курсором. Для того чтобы эти изменения приняли постоянный характер, с помощью менеджера ресурсов `xrdb` нужно откорректировать базы данных ресурсов X Window, что и выполняется второй командой:

```
# echo "XTerm*ttyModes:  erase ^?" | xrdb -merge
```

Выражение, используемое в команде `xmodmap`, может быть записано в файл. Сохраним выражение

```
keysym BackSpace = Delete
```

в файле xmodmap.ch:

```
echo keysym BackSpace = Delete > xmodmap.ch
```

Тогда для переприсвоения функций клавиш можно использовать командную строку:

```
cat xmodmap.ch | xmodmap -
```

11.3.3. Команда *xlsfonts*

Система X Window устанавливается с обширной базой шрифтов. Команда `xlsfonts` дает пользователю возможность получать информацию о шрифтах, изменять настройки и добавлять в систему новые шрифты. Чтобы отобразить, например, все шрифты размером 6×13, установленные в системе, необходимо выполнить команду:

```
# xlsfonts|grep 6x13
6x13
6x13
```

```
6x13bold
6x13bold
heb6x13
heb6x13
```

Можно установить для приложения те шрифты, которые более удобны. Например, при создании X-терминала можно установить для него шрифт 12×24:

```
# xterm -g 80x30+250+280 -fn `xlsfonts|grep 12x24 -m 1`
```

Приведенная здесь командная строка работает так, как описано далее. Терминальная сессия открывается с помощью команды `xterm`. В качестве параметров этой команды используются геометрические характеристики терминального окна (параметр `-g`). Для задания шрифта применяется опция `-fn`, при этом командой `xlsfonts` выводятся все шрифты 12×24. Команда `grep` выбирает первую строку с указанным шрифтом и передает ее в `xterm`.

11.4. Оконные менеджеры и графические оболочки

Напомню, что система X Window сама по себе не предоставляет клиенту готовый графический интерфейс для работы — X-сервер создает корневое окно, в котором могут запускаться простейшие программы-клиенты (`xedit`, `xterm` и т. д.). Система не указывает пользователю, что нужно делать, как создавать и работать с графическим интерфейсом.

Вместо этого X предоставляет пользователю инструменты, а он принимает решение, как их использовать. Такими инструментами являются оконные менеджеры и графические оболочки.

Фактически графический интерфейс пользователя в современных операционных системах можно изобразить тремя уровнями, каждый из которых представляет определенный уровень функциональности (рис. 11.5).

Самый первый уровень, наиболее близко находящийся к операционной системе, представлен X-сервером, принципы функционирования которого мы достаточно подробно проанализировали. На практике используется несколько наиболее известных дистрибутивов X-серверов: XFree86, Metro-X и Accelerated-X. X-сервер обычно устанавливается во время инсталляции операционной системы.

На втором уровне располагается оконный менеджер, который отвечает за управление окнами. Он представляет собой специальную клиентскую программу, которая указывает серверу, каким образом располагать окна, и обес-

печивает пользователю способ перемещения окон. В этом случае X-сервер служит посредником между пользователем и клиентом.

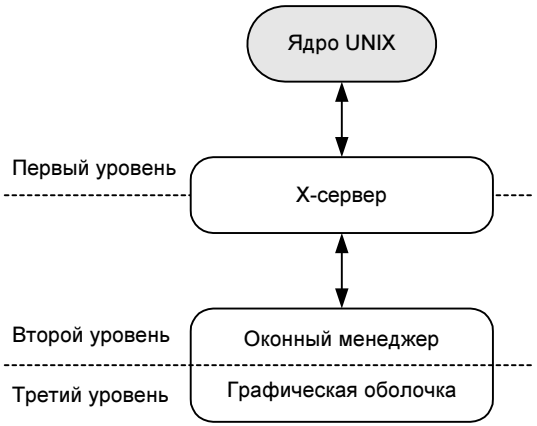


Рис. 11.5. Модель взаимодействия различных уровней графической системы

Границы окон, заголовок, размеры и т. д. — все эти параметры регулируются оконными менеджерами. Оконный менеджер не отвечает за содержимое окна — этим занимается графическая оболочка, представляющая собой третий уровень функциональности. Обычно оконный менеджер и графическая оболочка работают вместе, причем каждая графическая оболочка рассчитана на совместную работу с вполне определенными оконными менеджерами, хотя это не обязательно. Например, ранние версии графической оболочки GNOME работали с оконным менеджером Enlightenment, хотя более поздние версии этой же оболочки рассчитаны на работу с оконным менеджером Sawfish. Наиболее популярными оконными менеджерами являются WindowMaker, BlackBox, Ice WM, Sawfish, Enlightenment, AfterStep, Fvwm2 и Fvwm1.

В большинстве современных операционных систем предусмотрена возможность работы с различными графическими оболочками. Например, такая распространенная операционная система, как Linux, допускает работу с популярными графическими оболочками GNOME и KDE. Более того, поскольку все операционные системы UNIX являются многопользовательскими, то на одном компьютере может работать несколько графических оболочек, причем разные пользователи, работающие с одной и той же программой, могут использовать разные оболочки.

Рассмотрим более подробно особенности функционирования оконных менеджеров и графических оболочек.

Оконный менеджер не может корректно управлять окнами, не имея о них информации. Окна обладают различными свойствами, управление которыми должен обеспечивать именно менеджер окон. Например, во многих случаях необходимо отображать заголовки окон, в других желательно, чтобы окно имело фиксированный размер. Окно может быть свернуто в пиктограмму (значок) — в этом случае оконный менеджер должен знать, какую пиктограмму использовать и как она называется.

Менеджер получает от клиентов информацию о параметрах окна одним из способов:

- при создании X-окна ему могут быть переданы "подсказки" (hints) о начальных координатах окна, его геометрии, минимальных и максимальных размерах и т. д.;
- при использовании встроенного в X способа взаимодействия между программами — механизм "свойств".

Еще одной важной функцией, выполнение которой берет на себя оконный менеджер, является "политика фокусирования" мыши. Каждая оконная система должна иметь некоторый способ активизации окна при его выборе через нажатие клавиш, а также визуальную индикацию активности окна. Наиболее широко используется метод фокусировки (click-to-focus). Этот метод применяется в Microsoft Windows, и суть его в том, что окно становится активным после щелчка на нем мышью.

X не поддерживает какой-либо конкретный метод фокусирования — этим занимается оконный менеджер, который управляет передачей фокуса окну в определенные моменты времени. Различные оконные менеджеры поддерживают разные методы фокусирования. Все они поддерживают метод щелчка для фокусирования, и большинство из них — некоторые другие методы.

Самыми популярными методами фокусирования являются:

- focus-follows-mouse (фокус следует за мышью). Фокусом владеет окно, находящееся под указателем мыши. Оно не обязательно должно быть расположено поверх всех остальных окон. Фокус меняется при перемещении указателя на другое окно, при этом щелчок не требуется;
- sloppy-focus (нечеткий фокус). При использовании данного метода фокус меняется только тогда, когда курсор перемещается на другое окно, но не в момент ухода с текущего окна;
- click-to-focus (щелчок для выбора фокуса). Активное окно выбирается щелчком мыши, после чего окно может быть "поднято" и появиться поверх остальных окон. Все нажатия клавиш теперь будут направляться этому окну, даже если курсор мыши переместится к другому.

Многие оконные менеджеры поддерживают и другие методы, а также различные модификации вышеперечисленных. Современные оконные менеджеры предоставляют пользователю различные графические интерфейсы: одни поддерживают "виртуальные рабочие столы", другие позволяют изменять установки по умолчанию клавиш управления и навигации, третьи поддерживают несколько визуальных форм, позволяя тем самым изменять внешний вид.

Что же касается графических оболочек для X, то наиболее популярными из них являются KDE и GNOME. Они имеют интегрированные оконные менеджеры, причем каждый из них, как правило, обладает собственным механизмом настройки, а некоторые из них используют созданный вручную файл конфигурации.

В GNOME имеется панель для запуска приложений и отображения их состояния, а также рабочий стол с набором стандартных инструментов, на котором можно разместить данные и приложения. Кроме того, в графической оболочке предусмотрены механизмы взаимодействия приложений и пользователей, значительно облегчающие им работу.

Особенностью оболочки KDE является ее простота использования. Вот некоторые из большого числа возможностей, которые предоставляются пользователю данной оболочкой:

- ❒ современный рабочий стол;
- ❒ удобная справочная система;
- ❒ единый подход к управлению всеми приложениями оболочки;
- ❒ стандартизированные элементы управления (меню и панели инструментов), раскладки клавиатуры, цветовые схемы и т. д.;
- ❒ локализация более чем для 40 языков;
- ❒ диалоговый режим конфигурирования рабочего стола;
- ❒ возможность выбора большого количества полезных приложений для KDE.

Для KDE разработан пакет офисных приложений, включая программы для работы с электронными таблицами и создания презентаций, органайзер, клиент телеконференций и другие программы. В комплект поставки KDE входит также Web-браузер Konqueror, который является серьезной альтернативой другим обозревателям для UNIX-систем.

Вот как выглядят интерфейсы графических оболочек KDE (рис. 11.6) и GNOME (рис. 11.7).

Рассмотрим способы инсталляции и настройки графических оболочек в операционных системах UNIX на примере FreeBSD.

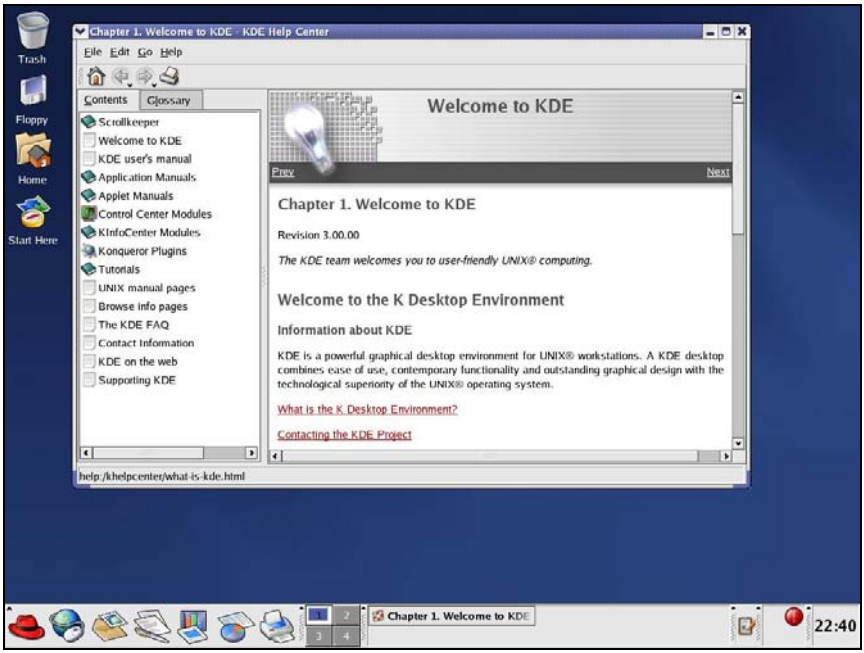


Рис. 11.6. Внешний вид графической оболочки KDE

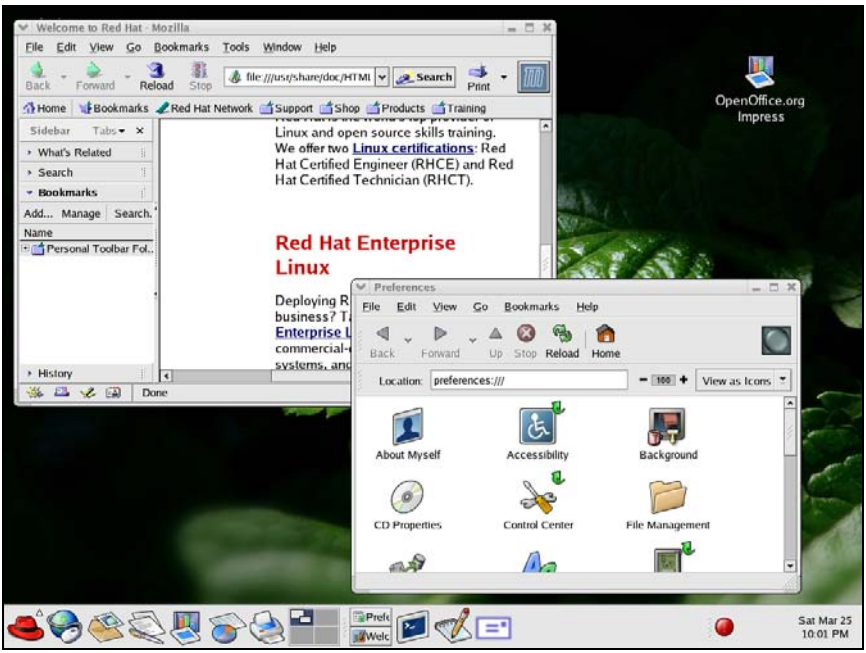


Рис. 11.7. Внешний вид графической оболочки GNOME

Начнем с GNOME. Наиболее легкий вариант установки этой оболочки — использование меню **Desktop Configuration**, которое появляется в процессе установки FreeBSD. Другим способом установки является инсталляция оболочки из пакета или коллекции портов, а также из исходных текстов (в этом случае используется дерево портов). Последовательность команд для установки из исходных текстов показана далее:

```
# cd /usr/ports/x11/gnome2
# make install clean
```

После установки GNOME необходимо указать X-серверу, что следует запускать оболочку GNOME вместо стандартного оконного менеджера. Для этого в файле .xinitrc нужно заменить строку (обычно это последняя строка), в которой указан оконный менеджер, той, что вызывает /usr/X11R6/bin/gnome-session. Вот как может выглядеть откорректированный файл .xinitrc:

```
#/etc/X11/xinit/.xinitrc
#!/bin/sh

xmodmap -e "keycode 22=BackSpace"
xsetroot -solid LightSlateGrey
xset s 600
xterm -g 80x20+150+8 &
xterm -g 80x20+150+325 &
xload -g +4+0 &
xclock -g +815+0 -digital &

# Запуск графической оболочки GNOME
# exec fvwm2    Эта строка должна быть закомментирована или удалена
/usr/X11R6/bin/gnome-session
```

Саму строку можно добавить в файл конфигурации с помощью команды:

```
# echo "/usr/X11R6/bin/gnome-session" > ~/.xinitrc
```

Если теперь выполнить команду `startx`, то будет запущена графическая оболочка GNOME. Здесь, однако, есть один нюанс. Если используется менеджер дисплея `xdm`, то GNOME не запустится. В этом случае нужно создать исполняемый файл .xsession, в котором будет присутствовать строка /usr/X11R6/bin/gnome-session.

Для этого следует отредактировать файл, заменив существующую команду запуска оконного менеджера на /usr/X11R6/bin/gnome-session:

```
# echo "#!/bin/sh" > ~/.xsession
```

```
# echo "/usr/X11R6/bin/gnome-session" >> ~/.xsession
# chmod +x ~/.xsession
```

Можно использовать еще один вариант — так настроить менеджер дисплея, чтобы он позволял выбирать оконный менеджер во время входа в систему.

Оболочку KDE, как и GNOME, проще всего установить через меню **Desktop Configuration** в процессе инсталляции FreeBSD. Хочу напомнить, что программное обеспечение легко установить из пакета или из коллекции портов. Для сборки KDE из исходных текстов можно воспользоваться деревом портов:

```
# cd /usr/ports/x11/kde3
# make install clean
```

Для запуска KDE по умолчанию нужно отредактировать файл .xinitrc так:

```
# echo "exec startkde" > ~/.xinitrc
```

Теперь при вызове X Window по команде `startx` в качестве оболочки будет использоваться KDE. При работе с менеджером дисплеев типа XDM настройка несколько отличается — отредактировать нужно файл .xsession. Дополнительную информацию по работе с оболочкой KDE можно получить из справочной системы. Неплохим справочным материалом по KDE является online-документация.

Глава 12

Разработка приложений в среде UNIX

Современные операционные системы UNIX имеют развитые средства разработки приложений, позволяя создавать как самые простые утилиты, работающие с текстовой консолью, так и многофункциональные приложения с графическим интерфейсом. Несмотря на такое разнообразие инструментов разработки, мы остановимся на двух из них — языках программирования C/C++ и Perl. Такой выбор обусловлен несколькими факторами.

Язык C/C++ является классическим инструментом разработки приложений для UNIX, впрочем, как и для других операционных систем. Помимо гибкости и мощи этого языка следует сказать, что сама операционная система написана на C/C++ и включает в себя многие библиотечные функции, что облегчает разработчику задачу создания приложений — достаточно уметь применять данные функции. Кроме того, ядро операционной системы использует механизм системных вызовов, посредством которых реализован API пользователя, позволяющий создавать приложения, очень тесно взаимодействующие с операционной системой.

Немаловажным преимуществом языка C/C++ является и то, что программы, разработанные для одной операционной системы, можно без особого труда модифицировать для работы в другой, т. е. C-код является легкопереносимым. При этом говорят о переносимости на уровне исходных текстов программ — исходный текст программы на C/C++, разработанной для какой-то одной UNIX-системы, можно с минимальными изменениями откомпилировать в другой. Более того, если в программе используются только функции стандарта POSIX и стандартные библиотеки C/C++, то никаких изменений в исходных текстах вообще не понадобится.

В этой главе мы рассмотрим наиболее широко распространенный компилятор языка C++, известный под аббревиатурой "g++", входящий в состав пакета свободно распространяемого программного обеспечения для разработки

приложений под названием GCC (GNU Compiler Collection). В состав дистрибутивного пакета программ GCC, кроме компилятора C++, входят компиляторы языков Java, Fortran и Ada.

Язык Perl (Practical Extraction and Report Language) был создан американским программистом Ларри Уоллом в начале 90-х годов прошлого века. Изначально Perl задумывался как высокоуровневый кроссплатформенный язык системного программирования. С тех пор этот язык вышел далеко за пределы своего исходного предназначения, но продолжает очень широко использоваться в системном программировании как для операционных систем UNIX, так и для других платформ. Для обеспечения максимальной переносимости основное внимание уделялось открытым системам, соответствующим стандарту POSIX, который поддерживает большинство UNIX-систем.

Программистов, использующих Perl, условно можно разделить на две категории: системных администраторов, создающих программы для управления файловой системой и процессами, поиска и составления отчетов, и пользователей, разрабатывающих электронные формы для Web-серверов. Последняя категория находит Perl более удобным и легким, чем C, поскольку Perl имеет развитые возможности для манипулирования данными, включая проверку данных и операции с простыми базами данных.

В настоящее время язык Perl является основным средством администрирования UNIX, который с успехом может использоваться вместо других традиционных средств администрирования. Интерпретатор этого языка, так же как и командный интерпретатор shell, в настоящее время является встраиваемым инструментальным средством во всех версиях операционных систем UNIX. Универсальность этого языка и способствовала его широкому распространению среди системных администраторов и программистов UNIX, учитывая и то, что он решает задачи обычно быстрее, чем другие аналогичные средства.

Следует подчеркнуть, что Perl может использоваться для системного программирования даже в системах, не соответствующих стандарту POSIX — в таких случаях вам понадобятся специализированные модули для этих систем.

Несмотря на признание языка Perl как инструмента системного администрирования, в последнее время он стал очень популярным в области разработки интернет-приложений, таких как CGI-сценарии, системы автоматической обработки электронной почты и поддержки Web-узлов.

Одной из широко применяемых в Интернете технологий является технология CGI-сценариев, с помощью которой реализуются динамические эффекты. Сценарии могут быть разработаны с помощью любого языка программирования, но написанные на Perl получили наибольшее распространение из-за легкости создания и оптимизационных возможностей этого языка при обра-

ботке текстовых файлов. Такая популярность привела к тому, что разработчики серверов Интернета, работающие в других операционных системах, стали включать возможность подключения сценариев Perl в свои системы. В настоящее время их можно использовать и на серверах Apache, NCSA и Netscape для операционной системы UNIX.

Следует отметить, что язык Perl очень часто применяется для решения задач, связанных с автоматизацией обработки электронной почты Интернета. Сценарии Perl можно использовать для фильтрации почты на основе адреса или содержимого, автоматического создания списков рассылки и для решения многих других задач. Можно, например, написать сценарий, который обрабатывает входящую почту и добавляет сообщения на заранее созданную страницу новостей, сортируя их по соответствующим тематикам, что позволяет быстро просматривать почту, не тратя время на чтение каждой полученной корреспонденции.

Еще одна сфера применения языка Perl — поддержка Web-узлов, представляющих собой структурированное хранилище HTML-страниц. Язык Perl оптимизирован для обработки большого количества текстовых файлов, поэтому его использование для анализа и автоматического изменения содержимого Web-узла само собой вытекает из тех задач, для решения которых он специально и создавался. Его, например, можно применять как для решения задачи проверки правильности перекрестных ссылок на страницах Web-узла, так и для проверки правильности ссылок на другие узлы.

Язык Perl обладает развитыми сетевыми возможностями, поэтому его легко можно использовать для обмена информацией на основе протоколов HTTP и FTP. Это позволяет автоматизировать получение файлов с других узлов, а в сочетании с его возможностями обработки текстовых файлов помогает создавать сложные информационные системы.

Рассмотрим некоторые важные практические аспекты программирования на языках C и Perl и начнем с C.

12.1. Разработка программ на C++

Разработку программы следует начать с ввода и редактирования исходного текста, для чего подойдет любой из популярных текстовых редакторов (`emacs`, `vi`, `vim` и т. д.). Затем исходный текст программы нужно сохранить в файле, имеющем расширение c или cpp, при этом учтите, что все имена в операционной системе UNIX чувствительны к регистру. Например, простейшую программу, отображающую приветствие на экране и имеющую исходный текст

```
#include <stdio.h>
```

```
void main(void)
{
  printf("Hello, world!\n");
}
```

можно сохранить в файле hello.cpp, после чего можно откомпилировать hello.cpp, получив при этом исполняемый файл программы. Для этого следует выполнить команду:

```
# g++ hello.cpp
```

По этой команде компилятор g++ попытается создать исполняемый файл из файла hello.cpp, содержащего исходный текст программы. Создание исполняемого файла программы требует выполнения, как минимум, двух шагов: на первом шаге файл с исходным текстом компилируется в объектный файл, а на втором из объектного файла создается исполняемый файл программы. Процесс сборки приложения иногда называют линковкой (от англ. *link* — связывать).

В простейшем случае, который здесь представлен, исполняемый файл создается из одного объектного файла, хотя в общем случае объектных файлов может быть несколько. На каждом из описанных этапов (компиляция и сборка) выполняются определенные задачи. На этапе компиляции из исходного текста программы генерируется объектный файл, представляющий собой двоичный код инструкций языка C в форме машинных команд процессора. Во время компиляции вылавливаются ошибки, связанные с неправильным использованием функций или с отсутствием их описания в файлах заголовков, а также различного рода синтаксические ошибки и т. д.

На втором этапе выполняется сборка программы. Здесь определяются взаимосвязи между различными частями программы, а также разрешаются все ссылки на вызываемые из разных объектных файлов функции и данные и т. д. Если не указано имя исполняемого файла, то после сборки ему автоматически присваивается имя a.out. Разработчик программы имеет возможность изменить это имя на более описательное. Если во время компиляции происходят какие-то ошибки, то компилятор g++ выводит соответствующие сообщения на экран дисплея, а исполняемый файл программы, естественно, не создается.

Если сборка программы прошла успешно, то ее можно запустить из командной строки, набрав

```
# ./a.out
```

Для компиляции программы и создания исполняемого файла с именем, отличным от a.out, следует задать опцию `-o`. Например, для создания исполняемого файла hello.out можно воспользоваться командой

```
# g++ -o hello.out hello.cpp
```

В результате компилятор создаст исполняемый файл hello.out.

Иногда требуется вместо двух этапов (компиляции и сборки) выполнить только компиляцию исходного текста. Результатом компиляции в этом случае будет объектный файл, представляющий программу в виде машинного кода, который затем можно связать с другим объектным файлом для сборки всей программы или использовать в другой программе.

Например, для осуществления только компиляции программы hello.cpp без создания исполняемого файла нужно выполнить команду

```
# g++ -c hello.cpp
```

В результате получим объектный файл hello.o (объектные файлы имеют расширение o).

Для создания исполняемого файла программы из нескольких файлов с исходными текстами нужно просто перечислить их все в командной строке, как в этом примере:

```
# g++ file1.cpp file2.cpp file3.cpp
```

Если компиляция прошла успешно, то будет создан один исполняемый файл a.out. Исполняемый файл программы можно создать из нескольких ранее созданных объектных файлов, например:

```
# g++ file1.o file2.o file3.o
```

После успешной сборки будет создан, как и в предыдущих случаях, исполняемый файл. Следующий пример демонстрирует создание исполняемого файла program.out из нескольких файлов с исходными текстами:

```
# g++ -o program.out file1.cpp file2.cpp file3.cpp
```

Здесь создается исполняемый файл program.out, использующий для сборки файлы file1.cpp, file2.cpp и file3.cpp с исходными текстами.

В процессе создания исполняемого файла, кроме файлов исходных текстов, могут понадобиться и файлы библиотечных функций или, по-другому, библиотеки, представляющие собой набор объектных файлов, сгруппированных вместе в один файл и проиндексированных. Библиотеки могут содержать определенный набор функций, как стандартных, так и определенных разработчиком. Библиотечные файлы позволяют тиражировать часто применяемые функции для использования их в других приложениях.

Присоединяемые к приложению библиотеки можно задать в командной строке в форме `-l`*имя_библиотеки*. Например, `-lg++` указывает на библиотеку стандартных функций C++, а `-lm` — на библиотеку, содержащую различные математические функции (синус, косинус, арктангенс и т. д.). Все библиотеки

должны быть перечислены в командной строке после объектных файлов или файлов исходных текстов, содержащих вызовы библиотечных функций.

Хочу более подробно остановиться на некоторых, очень важных аспектах программирования. Первый касается разработки сетевых приложений в среде UNIX. В некоторых операционных системах, например, Linux и FreeBSD, сетевые функции включены в ядро и вызываются посредством системных вызовов (`socket()`, `bind()`, `connect()`, `listen()`, `accept()`). В таких случаях компиляция и сборка сетевых приложений выполняется, как обычно:

```
# g++ -o program file.cpp
```

Здесь `file.cpp` — файл с исходным текстом, а `program` — исполняемый файл.

Если сетевое приложение разрабатывается, например, в операционной системе Solaris, то здесь ситуация иная. Сетевые функции реализованы не в ядре, а в виде библиотечных функций C++, поэтому командная строка компилятора должна выглядеть так:

```
g++ -o program file.cpp -lsocket -lnsl library
```

Здесь `program` — исполняемый файл, `file.cpp` — файл с исходным текстом, `library` — библиотека сетевых функций.

Кроме того, в исходный текст программы должны быть включены строки

```
#include <sys/types.h>
#include <sys/socket.h>
```

Например, в операционной системе Solaris 10 командная строка может выглядеть так:

```
# g++ -o program file.cpp -lsocket -lnsl /usr/lib/libsocket.so
```

Здесь для создания сетевого приложения `program`, исходный текст которого находится в `file.cpp`, используется библиотека `libsocket.so` (или `libsocket.so.1`).

Второй момент связан с разработкой приложений, использующих программные потоки. Для того чтобы программа была успешно создана, при сборке следует указать на необходимость использования специальной библиотеки, в которой реализованы потоковые функции.

Например, в операционной системе Solaris командная строка для сборки приложения, использующего потоковые функции, будет выглядеть следующим образом:

```
# g++ -o program file.cpp -lpthread /lib/libthread.so
```

В самом исходном тексте программы должно присутствовать объявление файла заголовка:

```
#include <pthread.h>
```

Для операционных систем Linux командная строка может выглядеть так:

```
# g++ -o program file.cpp -lpthread
```

Сложные программы, разрабатываемые на C/C++, могут содержать несколько файлов, включая как файлы с исходными текстами программ (расширение cpp), так и файлы заголовков (расширение h), в которых размещено описание функций. Кроме того, для сборки всего приложения может понадобиться определенная последовательность действий, которую очень сложно, а иногда и невозможно выразить командной строкой. В таких случаях сборка приложения выполняется посредством так называемого make-файла, содержащего перечень правил и взаимозависимостей между командами, выполняющими создание исполняемых файлов. Содержимое make-файла обрабатывается утилитой `make`.

Использование make-файла рассмотрим на следующем примере. Предположим, что для сборки приложения необходимо использовать файлы, перечисленные далее:

```
main1.cpp
mylib.cpp
mylib.h
openfile.cpp
openfile.h
```

Наш исполняемый файл назовем program.out. Для создания make-файла (назовем его program.make) можно воспользоваться одним из редакторов (`vi`, `emacs` и т. д.). Если для создания приложений использовать отдельный каталог, то можно присвоить make-файлу имя "makefile".

Вот как может выглядеть make-файл с именем program.make:

```
# program.make – это комментарий, и он игнорируется утилитой make
#
program.out : main1.o mylib.o openfile.o
       g++ -o program.out main1.o mylib.o openfile.o

# Из двух инструкций, представленных выше, видно, что program.out зависит
# от объектных файлов main1.o, mylib.o и openfile.o и должна из них
# компоноваться.
# Для создания program.out нужно выполнить команду g++ с необходимыми
```

```
# аргументами (вторая строка). Таким образом, исполняемый файл
# program.out создается из трех объектных файлов.

main1.o: main1.cpp openfile.h mylib.h
    g++ -c main1.cpp

# В этих двух инструкциях определяется, как создавать объектный файл
# main1.o (первая инструкция), и указана командная строка для компиляции
# (вторая строка). При этом компиляция выполняется только в том случае,
# если хотя бы один из файлов main1.cpp, openfile.h или mylib.h изменился
# со времени последней компиляции main1.o.

mylib.o: mylib.cpp mylib.h
    g++ -c mylib.cpp

# Инструкции, показанные выше, указывают на то, что файл mylib.o зависит
# от mylib.cpp и mylib.h (первая инструкция) и должен компилироваться в
# соответствии с командной строкой, указанной во второй инструкции. При
# этом компиляция выполняется в том случае, если хотя бы один из файлов
# mylib.cpp или mylib.h изменился со времени последней компиляции файла
# mylib.o

openfile.o: openfile.cpp openfile.h
       g++ -c openfile.cpp

# Эти инструкции показывают зависимость файла openfile.o от openfile.cpp
# и openfile.h (первая строка) и предписывают откомпилировать файл
# openfile.cpp, если файлы openfile.cpp и openfile.h изменились со
# времени последней компиляции openfile.o

clean:
  rm *.o program.out

# Здесь последние две инструкции указывают правила очистки рабочего
# каталога приложения от ненужных файлов (например, при необходимости
# полной перекомпиляции приложения). Из второй инструкции видно, что в
# процессе очистки должны быть удалены все объектные файлы и исполняемый
# файл.
```

После формирования make-файла для создания приложения можно выполнить командную строку

```
# make -f program.make
```

Если все объектные файлы, указанные в make-файле, были созданы успешно, то в результате получим программу program.out.

Выполнить программу можно стандартным образом:

```
# ./program.out
```

Следует отметить, что если хотя бы один из файлов (с расширением cpp или h) изменился, то нужно опять выполнить командную строку

```
# make -f program.make
```

Если нужно полностью перекомпилировать приложение, то следует выполнить команду

```
# make -f program.make clean
```

Таким образом, процесс создания исполняемого файла программы с помощью make-файла и утилиты `make` можно представить последовательностью шагов:

- `# emacs program.make` — создаем файл program.make с помощью редактора `emacs` (можно использовать и другой редактор, например, `vi`);
- `# make -f program.make` — создаем программу program.out.

Если для разработки приложения используется отдельный каталог, то можно использовать make-файл с именем makefile. Тогда команды для создания исполняемого файла несколько изменятся:

- `# emacs makefile` — создаем файл с именем makefile с помощью редактора `emacs`;
- `# make` — создаем программу program.out (утилита `make`, заданная без параметров, использует по умолчанию файл makefile).

Для очистки рабочего каталога от ненужных объектных файлов следует выполнить команду

```
# make clean
```

Вот еще один пример использования make-файлов для создания приложений:

```
# account.make — это комментарий, и он игнорируется утилитой make.
# В этом примере создается исполняемый файл account. Для сборки
# приложения используются следующие файлы:
#   - исходных текстов: account.cpp и accountmain.cpp;
```

```
#   - заголовков: account.h.
#
account: account.o accountmain.o
       g++ -o account account.o accountmain.o

account.o: account.cpp account.h
       g++ -c account.cpp

accountmain.o: accountmain.cpp account.h
       g++ -c accountmain.cpp

clean:
       rm account account.o accountmain.o
```

На этом рассмотрение возможностей компилятора g++ можно закончить. Дополнительную информацию о компиляторе g++ можно найти по адресу **http://gcc.gnu.org/onlinedocs**.

12.2. Perl

Прежде чем приступить к изучению языка Perl, хочу продемонстрировать его возможности. Многие программы, написанные на этом языке, занимают всего несколько строк, выполняя те же функции, что и сложные командные файлы, в которых задействованы команды `find`, `awk`, `sed` и т. д.

Вот простой пример. Предположим, имеются три файла — 1.tst, 2.tst и 3.tst со следующим содержимым:

```
$ cat 1.tst
1234567890
$ cat 2.tst
0123456789
$ cat 3.tst
0123006789
```

Пусть требуется заменить цифру 0 в этих файлах на символ N. Это легко выполняется при помощи команды

```
$ perl -e 's/0/N/gi' -p -i *.tst
```

После выполнения операции содержимое данных трех файлов будет следующим:

```
$ cat 1.tst
123456789N
```

```
$ cat 2.tst
N123456789
$ cat 3.tst
N123NN6789
```

Рассмотрим более сложный пример. Пусть файл `1.tst` содержит следующие строки:

```
# cat 1.tst
111 2222 7777
VR 33LG 9037
9012 FT 34
WE JK355 133
```

Требуется заменить первые два элемента в строках, начинающихся двумя цифрами, на строку `NN`. Вот как можно это реализовать при помощи Perl:

```
# perl -e 's/^\d{2}/NN/gi' -p -i 1.tst
```

Результат преобразования содержимого файла выглядит так:

```
# cat 1.tst
NN1 2222 7777
VR 33LG 9037
NN12 FT 34
WE JK355 133
```

Как видно даже из этих примеров, далеко не самых сложных, язык Perl обладает весьма широкими возможностями по обработке данных.

Еще большего можно добиться, разрабатывая полноценные приложения в классическом стиле с использованием всех функций этого языка. Вот как можно модифицировать предыдущий пример (исполняемый файл называется replace_demo.pl):

```
#!/usr/bin/perl -w

$orig='^\d{2}';
$replacement='NN';
$newfile = "new";
foreach $file (@ARGV)
{
  if (! open(INPUT,"<$file") )
  {
    print "Can't open input file $file\n";
    next;
```

```
  }

  open(OUTPUT, "+>>$newfile");
  select(OUTPUT);
  while (<INPUT>)
  {
    $data = $_;
    $data =~ s/$orig/$replacement/gi;
    print  OUTPUT $data;
  }
  close (INPUT);
  close(OUTPUT);

  unlink("$file");
  rename("new", $file) or die "Can't rename new to $file: $!";
}
exit(0);
```

Здесь я не буду объяснять смысл операторов Perl, далее мы более подробно остановимся на этом. Замечу, что данная программа может принимать несколько аргументов командной строки (`$ARGV[0]`, `$ARGV[1]`, `$ARGV[2]`), в качестве которых должны выступать имена файлов.

Следует сказать, что программы на языке Perl легко пишутся, но не всегда легко читаются, учитывая в особенности то, что выполнить одну и ту же операцию можно несколькими способами.

Например, операторы, представленные далее, являются эквивалентными:

```
if ($x == 0) {$y = 10;}  else {$y = 20;}
$y = $x==0 ? 10 : 20;
$y = 20;  $y = 10 if $x==0;
```

Далее мы рассмотрим язык Perl более подробно и начнем с запуска программы.

12.2.1. Запуск программ на языке Perl

Запустить программу, написанную на Perl, не очень сложно. Рассмотрим простой пример запуска программы (назовем ее hello.pl), исходный текст которой представлен далее:

```
#!/usr/bin/perl -w
if ($#ARGV >= 0)
{
```

```
  $who = join(' ', @ARGV);
}
else
{
 $who = 'World';
}
print "Hello, $who!\n";
```

Здесь первая строка, которая начинается с символов `#!`, определяет расположение программы Perl (обычно это каталог /usr/bin, хотя может быть и другой). В этой же строке параметр `-w` указывает на необходимость вывода предупреждающих сообщений.

Сам текст программы несложен. Если задан параметр командной строки, например, `root`, то программа выводит сообщение

```
Hello, root!
```

Если параметр не указан, то будет выведено сообщение

```
Hello, World!
```

После создания файла hello.pl нужно проверить атрибуты доступа — файл должен иметь доступ по чтению/записи и выполнению. Если какие-либо атрибуты отсутствуют, то, выполнив команду `chmod`, можно их установить. Например, чтобы владелец файла мог его прочитать, изменить и выполнить, нужно набрать командную строку

```
# chmod 700 hello.pl
```

Для того чтобы файл можно было прочитать и выполнить всем остальным пользователям, наберите команду:

```
# chmod a+rx hello
```

Сама программа после этого запускается, как обычно, например:

```
# ./hello root
```

Стандартные дистрибутивы языка Perl включают огромное число функций, позволяющих выполнять большинство операций по обработке данных. Тем не менее, в целом ряде случаев требуется использовать функции, определенные в отдельных программных модулях. Для того чтобы сделать такие функции доступными, следует указать модуль, в который они включены, при помощи директивы `use`:

```
#!/usr/bin/perl -w
use Модуль;
. . .
```

12.2.2. Скалярные переменные и массивы

Приступая к знакомству с любым языком программирования, следует, прежде всего, выяснить, а какие типы данных он позволяет обрабатывать. Данные, в свою очередь, определяются своим типом — множеством значений и набором допустимых операций. В языке Perl можно использовать все буквы латинского алфавита (прописные и строчные), арабские цифры и символ подчеркивания _. Perl относится к языкам, чувствительным к регистру, — это означает, что символы прописной и строчной буквы считаются различными. Таким образом, два идентификатора `Str` и `str` задают две разные переменные.

Базовой единицей для работы с данными в Perl является скаляр (scalar) — отдельное значение, хранящееся в отдельной (скалярной) переменной. В скалярных переменных можно хранить строки, числа и ссылки. При этом строки могут иметь произвольную длину и содержать произвольные данные, включая двоичные последовательности с завершающим нулем. Числовые значения обычно хранятся в формате вещественных чисел с двойной точностью. Что же касается ссылок, то в них хранятся адреса переменных.

Скалярная величина может быть определенной или неопределенной. Определенная величина может содержать, как уже было сказано, число, строку или ссылку, при этом определенными величинами являются, например, 0 и пустая строка. Неопределенная величина может принимать единственное значение `undef`, для проверки которого можно воспользоваться функцией `defined`.

Все переменные должны начинаться с символа `$`, например:

```
$a=1;
$b="string";
```

Здесь переменной `$a` присваивается числовое значение, равное 1, а переменной `$b` — символьная строка `string`. Хочу отметить важный момент: интерпретатор Perl автоматически определяет тип переменной в момент присваивания ей значения, поэтому переменная `$a` в нашем примере будет считаться целочисленной, а переменная `$b` — строковой. Если требуется преобразовать строковое значение в число, то можно использовать POSIX-функцию `strtod`.

В языке Perl существует несколько переменных, имеющих специальные назначения. К ним относится особый тип переменной, которая обозначается, как `$_`, и называется переменной по умолчанию. Эта переменная может принимать, например, строку со стандартного ввода или файла, если явным образом не указана переменная-приемник. Переменная `$!`, если используется в числовом контексте, содержит текущее значение номера ошибки. Эта же переменная, используемая в строковом контексте, содержит соответствующее системное сообщение об ошибке. Применение этой переменной будет пока-

зано в последующих примерах программного кода. Далее приводится смысл еще нескольких специальных переменных:

- `$$` — идентификатор текущего процесса;
- `$0` — имя файла, в котором находится выполняемая программа;
- `$ARGV` — имя текущего файла, из которого выполняется чтение данных;
- `@ARGV` — массив аргументов командной строки, переданных программе.

Подобно другим языкам программирования, Perl позволяет выполнять над переменными различные арифметические и логические операции, что продемонстрировано в двух примерах далее.

Следующая программа иллюстрирует принципы выполнения арифметических операций.

```
#!/usr/bin/perl -w
$a = 39;
$b = -99;
$sum = $a+$b;
$sub = $a-$b;
$mul = $a*$b;
$div = $a/$b;
$remainder = $b % $a;
#
print "Sum = $sum\n";
print "Sub = $sub\n";
print "Mul = $mul\n";
print "Div = $div\n";
print "Remainder = $remainder\n";
```

Результат работы программы (она называется arithmetic.pl) будет выведен на экран консоли:

```
# ./arithmetic.pl
Sum = -60
Sub = 138
Mul = -3861
Div = -0.393
Remainder = 18
```

Далее показан пример выполнения операций сравнения:

```
#!/usr/bin/perl -w

print "Enter first number:";
```

```
$a = <STDIN>;
print "Enter second number:";
$b = <STDIN>;
if ($a > $b)
{
 print ("Max = $a\n");
}
if ($a < $b)
{
 print ("Max = $b\n");
}
 if ($a == $b)
{
 printf("Numbers are equal.\n");
}
```

Здесь показано выполнение операций сравнения чисел на "больше/меньше" или "равно", при этом на экране дисплея отображается соответствующий результат:

```
# ./logical.pl
Enter first number:-76
Enter second number:-762
Max = -76
```

Переменные `$a` и `$b` получают значения из устройства стандартного ввода, которое в языке Perl определено как `STDIN`:

```
$a = <STDIN>;
$b = <STDIN>;
```

Кроме того, в этом примере показано применение оператора условия `if`. Собственно, синтаксис условного оператора `if` языка Perl практически ничем не отличается от аналогичного оператора в C или других языках высокого уровня, поэтому останавливаться на нем подробно я не буду. То же самое касается и оператора цикла `while`, и других логических конструкций — более подробно они будут рассмотрены далее.

Программе может понадобиться вычислять не только простейшие соотношения "равно/не равно" или "больше/меньше", но и более сложные выражения. Далее приводится пример программы, выводящей на экран дисплея число, введенное с консоли, только в том случае, если оно находится в диапазоне от –100 до +100:

```
#!/usr/bin/perl -w
```

```
print "Enter number between -100 and 100:";
$a = <STDIN>;
if (($a > -100) && ($a < 100))
{
 print ("Number = $a\n");
}
if (($a <= -100) || ($a >= 100))
{
 print "Number is out of range";
}
```

В этом примере в операторах `if` указаны два условия:

```
($a > -100) && ($a < 100)
($a <= -100) || ($a >= 100)
```

Первое условие будет истинно только в том случае, если переменная `$a` будет больше –100 и меньше 100. Для истинности второго условия достаточно выполнения одного из условий: либо переменная `$a` меньше 100, либо больше 100.

Кроме простых скалярных переменных, Perl позволяет оперировать и с группами элементов, представленными в виде массивов и хэшей. Здесь следует отметить, что строки не считаются массивом байтов, поэтому к отдельному символу нельзя обратиться по индексу, как к элементу массива — для этого следует использовать функцию `substr`.

Для обозначения массива используется символ `@` перед именем массива, например:

```
@num_array= (0, 2, 4, 6)
```

Это выражение определяет массив `num_array` из четырех четных чисел. В следующем примере (исполняемый файл 1.pl) на экране дисплея отображается массив из четырех вещественных чисел, разделенных пробелами:

```
#!/usr/bin/perl -w
@array = (12.98, -56.02, 76.122, -4.031);
print "@array ";
```

Результат работы программы:

```
# ./1.pl
12.98 -56.02 76.122 -4.031
```

Тот же самый результат можно получить, запустив программу с оператором `foreach`:

```
#!/usr/bin/perl -w
```

```
@array = (12.98, -56.02, 76.122, -4.031);
print "Array: ";
print "$_ " foreach (@array);
```

Для обращения к отдельному элементу массива следует использовать индекс, а само обращение должно записываться в форме *$имя_массива[индекс]*. Например, для вывода на консоль 3-го элемента массива можно воспользоваться программным кодом, представленным далее:

```
#!/usr/bin/perl -w
@array = (12.98, -56.02, 76.122, -4.031);
print "$array[2]\n";
```

Вот как выглядит результат работы этой программы (она называется 2.pl):

```
# ./2.pl
76.122
```

Если необходимо вывести несколько элементов массива, не обязательно идущих подряд, можно воспользоваться следующим программным кодом:

```
#!/usr/bin/perl -w
@array = (12.98, -56.02, 76.122, -4.031);
print "Array: ";
print "@array[1,3] ";
```

Здесь на экран выводятся 2-й и 4-й элементы массива `@array`.

С элементами массивов можно выполнять целый ряд операций. Например, для присвоения отдельным элементам массива определенных значений можно воспользоваться специальной формой оператора присваивания, как в этом примере:

```
#!/usr/bin/perl -w
@array = (0, 0, 0, 0, 0, 0, 0);
@array[1, 3, 5] = (-11, -19, -39);
print "@array ";
```

Результат работы программы будет таким:

```
0 -11 0 -19 0 -39 0
```

Если необходимо добавить элементы в конец массива, то можно сделать так, как показано в примере:

```
#!/usr/bin/perl -w
@array = (0, 0, 0, 0, 0, 0, 0);
@array[1, 3, 5] = (-11, -19, -39);
@array = (@array, 99, 77);
print "@array ";
```

Вывод программы показан далее:

```
0 -11 0 -19 0 -39 0 99 77
```

Для добавления элементов в конец массива часто бывает удобно воспользоваться функцией `push`. Предыдущий пример, переписанный с помощью `push`, будет выглядеть так:

```
#!/usr/bin/perl -w
@array = (0, 0, 0, 0, 0, 0, 0);
@array[1, 3, 5] = (-11, -19, -39);
push(@array, 99, 77);
print "@array ";
```

Чтобы удалить последний элемент массива, следует применить команду `pop`. Следующий пример (он называется push_pop_demo.pl) демонстрирует это:

```
#!/usr/bin/perl -w
@array = (0, 0, 0, 0, 0, 0, 0);
@array[1, 3, 5] = (-11, -19, -39);
push(@array, 99, 77);
print "After push: @array \n";
$a1 = pop(@array);
$a2 = pop(@array);
print "After pop: @array \n";
print "\$a1 = $a1, \$a2 = $a2";
```

Обратите внимание на последнюю строку программы — здесь для вывода на экран специального символа `$` необходимо ликвидировать его специальное значение при помощи символа обратного слэша `\`.

Программа выводит на экран такой результат:

```
$ ./push_pop_demo.pl
After push: 0 -11 0 -19 0 -39 0 99 77
After pop: 0 -11 0 -19 0 -39 0
$a1 = 77, $a2 = 99
```

Весьма полезными являются и функции `shift` и `unshift`. Функция `shift` удаляет первый элемент массива (с индексом 0), а `unshift` — добавляет первый элемент в массив. Следующий пример (он называется shift_unshift_demo.pl) иллюстрирует это:

```
#!/usr/bin/perl -w
@array = (0, 0, 0, 0, 0, 0, 0);
@array[1, 3, 5] = (-11, -19, -39);
```

```
print "@array \n";
unshift(@array, 999);
print "After unshift: @array \n";
shift(@array);
shift(@array);
print "After shift: @array \n";
```

Вот результат выполнения программы:

```
$ ./shift_unshift_demo.pl
0 -11 0 -19 0 -39 0
After unshift: 999 0 -11 0 -19 0 -39 0
After shift: -11 0 -19 0 -39 0
```

Количество элементов в массиве можно определить, используя выражение `$#имя_массива+1`. Если массив не определен, то вычисленное выражение будет равно 0. Вот простой пример (он называется sarray.pl):

```
#!/usr/bin/perl -w
@array = (5, -7, 25, 1, -9, 7, -3);
$sizeof_array = $#array+1;
print "Size of array = $sizeof_array\n";
```

Программа выводит на консоль такой результат:

```
$ ./sarray.pl
Size of array = 7
```

12.2.3. Хэши

Хэш представляет собой массив, состоящий из пар "ключ-значение", при этом доступ к каждому значению записи осуществляется по ассоциированному с ним ключу. Хэш можно представить как `%имя_хэша`, а доступ к его отдельным элементам выполняется в форме `$имя_хэша{выражение}`.

Вот примеры хэшей:

```
$hash{1}="key1";
$hash{'myset'}="file.txt";
```

Хэш должен содержать четное количество элементов.

Для удаления элементов из хэша нужно воспользоваться функцией `delete`:

```
delete($hash{1});
```

Чтобы выделить отдельные ключи и значения хэша, нужно воспользоваться функциями `keys()` и `values()` соответственно. Например, для хэша `%hash(1,5,2,6,3,7)` выполнение этих функций дает следующий результат:

```
@key = keys(%hash);
```

После выполнения функции `keys()` переменная `@key` будет содержать значения (1, 2, 3).

```
@value = values(%hash);
```

После выполнения функции `values()` переменная `@value` будет содержать значения (5, 6, 7).

12.2.4. Операции и выражения

Язык Perl обеспечивает обработку данных с помощью некоторого набора операций — определенных действий над операндами, результатом которых является новое значение. Все операции можно разделить на несколько основных типов, часть из которых перечислена далее.

Бинарные арифметические операции включают четыре арифметических действия: сложение (`+`), вычитание (`-`), умножение (`*`) и деление (`/`), а также остаток от деления двух целых чисел (`%`) и возведение в степень (`**`). Примененные к числовым или строковым данным, которые содержат правильные литералы десятичных чисел, они выполняют соответствующие арифметические действия.

Логические операции включают бинарные операции логического сложения ИЛИ (`||`), логического умножения И (`&&`) и унарную операцию логического отрицания `!`. Результат операции `||` (логическое ИЛИ) является истинным, если истинен хотя бы один из операндов, в остальных случаях он является ложным. Операция логического И (`&&`) возвращает в качестве результата истину, только если оба операнда истинны, в противном случае ее результат — ложь. Операция логического отрицания `!` работает как переключатель: если ее операнд истинен, то она возвращает ложь, если операнд имеет значение ложь, то ее результатом будет истина.

Операции отношения используются для сравнения скалярных данных или значений скалярных переменных. Язык Perl предлагает набор бинарных операций, вычисляющих отношения равенства, больше, больше или равно и т. д. между операндами, поэтому эту группу операций еще называют операциями отношения. Следует отметить, что для сравнения числовых и строковых данных Perl использует разные операции. Все они представлены в табл. 12.1.

Операция присваивания обозначается, как `=`, и является бинарной операцией, правый операнд которой может быть любым правильным выражением, тогда как левый операнд должен определять область памяти, куда помещается вычисленное значение правого операнда. В качестве левого операнда операции присваивания можно использовать переменную любого типа или элемент массива. В предыдущих примерах мы уже сталкивались с операцией присваивания.

Таблица 12.1. *Операции отношения*

Операция	Числовая	Строковая	Значение
Равенство	==	eq	Истина, если операнды равны, иначе ложь
Неравенство	!=	ne	Истина, если операнды не равны, иначе ложь
Меньше	<	lt	Истина, если левый операнд меньше правого, иначе ложь
Больше	>	gt	Истина, если левый операнд больше правого, иначе ложь
Меньше или равно	<=	le	Истина, если левый операнд меньше правого или равен ему, иначе ложь
Больше или равно	>=	ge	Истина, если левый операнд больше правого или равен ему, иначе ложь
Сравнение	<=>	cmp	0, если операнды равны; 1, если левый операнд больше правого; –1, если правый операнд больше левого

В следующем примере

```
$a = $b + 10;
```

вычисляется значение правого операнда и присваивается переменной `$a`.

В языке Perl допускается осуществлять присваивание одного и того же значения нескольким переменным в одном выражении, например:

```
$var1 = $var2 = $var1[0] = 1;
```

При реализации вычислительных алгоритмов приходится определять значения некоторой переменной, после чего результат присваивать этой же переменной. Например, если увеличить на 5 значение переменной `$a`, и результат присвоить этой же переменной, то эту операцию можно реализовать так:

```
$a = $a+5;
```

Для таких и других подобных случаев в языке Perl предусмотрен более эффективный способ решения при помощи оператора составного присваивания `+=`, который прибавляет к значению левого операнда значение правого операнда и результат присваивает переменной, представленной левым операндом, например:

```
$a += 7;
```

Этот оператор эквивалентен оператору

```
$a = $a+7;
```

Операция составного присваивания более эффективна, чем обычное присваивание, поскольку в составном операторе присваивания переменная `$a` вычисляется один раз, тогда как в простом операторе ее приходится вычислять дважды.

Операции над строками включают операции конкатенации (соединения) двух строковых операндов и операции повторения, которые объединяют два строковых операнда в одну строку.

Операция конкатенации обозначается символом точка (`.`). Пример конкатенации строк:

```
"first_string"."second_string";
```

В результате получится `"first_stringsecond_string"`.

Результирующая строка содержит оба операнда, причем пробел между первым и вторым операндом отсутствует. Эта операция используется для присваивания переменной некоторого нового значения.

Для конкатенации двух и более строк через пробелы следует воспользоваться функцией `join`.

Вот еще один пример соединения строк:

```
$sl = "First";
$s2 = "Second";
$s3 = "Third";
$s = $sl.$s2.$s3;
```

В результате строка `$s` будет содержать значение `"FirstSecondThird"`.

Операция повторения строки обозначается символом `x`. С помощью этой операции создается новая строка, повторяющая строку, заданную левым операндом, указанное число раз. Вот пример операции повторения:

```
"AB" x 2;
```

Результатом является строка `"ABAB"`.

Операцию повторения удобно использовать при повторении массивов для заполнения их одинаковыми значениями, например:

```
@array = ("Y") x 3;
```

В результате массив `@array` будет содержать элементы (`"Y"`, `"Y"`, `"Y"`).

Символ операции повторения `x` следует отделять пробелами от операндов, т. к. иначе он может быть воспринят интерпретатором, как относящийся

к лексеме, а не представляющий операцию повторения. Например, интерпретатор Perl воспринимает строку xxy как состоящую из двух переменных $xx и $y, поэтому правильной записью будет $x x $y.

12.2.5. Логические структуры Perl

Perl, как и другие языки высокого уровня, включает в себя много логических структур, позволяющих изменить линейное (последовательное) выполнение операторов программы и организовать вычислительные алгоритмы, зависящие от результатов выполнения операций. Рассмотрим наиболее важные из этих структур и начнем с оператора if. С ним мы уже сталкивались ранее, теперь рассмотрим его более подробно.

Оператор if можно представить в одной из двух форм:

```
if (условие) {
 операторы;
}
```

или

```
оператор if условие;
```

Вот пример короткой программы с оператором if:

```
#!/usr/bin/perl -w
$data = <STDIN>;
chomp($data);
print "You entered: $data" if ($data eq "OK");
```

Здесь переменной $data присваивается строка, полученная со стандартного ввода STDIN, затем функция chomp убирает символ возврата каретки, после чего значение переменной $data выводится на экран, если оно равно OK.

Альтернативным к оператору if является unless (если не). Вот как будет выглядеть модифицированный вариант предыдущего примера, если вместо if используется unless:

```
#!/usr/bin/perl -w
$data = <STDIN>;
chomp($data);
print "You entered: $data" unless ($data ne "OK");
```

Следующая логическая структура, которую мы рассмотрим, является оператором цикла while. В общем виде синтаксис оператора можно представить так:

```
while (условие) {
```

```
  операторы;
}
```

Здесь *условие* представляет собой выражение, значение которого проверяется в начале каждой итерации. Операторы цикла выполняются до тех пор, пока *условие* является истинным. Если выражение становится ложным, происходит выход из цикла. Рассмотрим следующий пример:

```
#!/usr/bin/perl -w
@array = ("one", "two", "three", "four", "five", "six");
$index = 0;
while ($array[$index] ne "five")
{
  print "$array[$index]\n";
  $index += 1;
}
```

Здесь на экран дисплея выводятся все элементы массива `@array` вплоть до строки `"five"`. Условием выполнения цикла является неравенство текущего элемента значению `"five"`.

Вместо `while` можно использовать оператор `until`. В этом случае предыдущий пример можно модифицировать следующим образом:

```
#!/usr/bin/perl -w
@array = ("one", "two", "three", "four", "five", "six");
$index = 0;
until ($array[$index] eq "five")
{
  print "$array[$index]\n";
  $index += 1;
}
```

Во многих случаях требуется выполнить определенное число повторяющихся операций. В языке Perl предусмотрена такая возможность, и реализована она посредством оператора `for`. Функционирование этого оператора лучше всего продемонстрировать на примере:

```
#!/usr/bin/perl -w
@array = (5, -7, 25, 1, -9, 7, -3);
$sizeof_array = $#array+1;
print "Size of array = $sizeof_array\n";
$sum = 0;
for ($index = 0; $index < $sizeof_array; $index++)
```

```
{
  $sum += $array[$index];
}
print "Sum of elements = $sum\n";
```

Результат выполнения программы:

```
$ ./for_demo1.pl
Size of array = 7
Sum of elements = 19
```

Последняя логическая структура, которая будет рассмотрена, реализована в виде оператора `foreach`, специально предназначенного для обработки массивов. Оператор `foreach` можно представить следующим образом:

```
foreach $переменная (@массив) {
 операторы
}
```

Вот простейший пример применения оператора `foreach`:

```
#!/usr/bin/perl -w
@array = (5, -7, 25, 1, -9, 7, -3);
foreach $entity (@array)
{
  print "$entity ";
}
```

Здесь на экран консоли выводится список элементов массива `@array`. Можно упростить исходный текст предыдущего примера, если воспользоваться переменной по умолчанию `$_`:

```
#!/usr/bin/perl -w
@array = (5, -7, 25, 1, -9, 7, -3);
foreach (@array)
{
  print "$_ ";
}
```

12.2.6. Регулярные выражения

При обработке массивов данных и текстовых файлов часто возникает задача поиска элементов, удовлетворяющих заданным условиям, и выполнения над ними определенных операций. Условия поиска можно выразить посредством так называемых регулярных выражений, позволяющих описать образец или

шаблон поиска при помощи специальных правил, а манипуляции с регулярными выражениями осуществляются при помощи соответствующих операций.

Регулярное выражение представляет собой набор правил для описания текстовых строк, а сами правила записываются в виде последовательности обычных символов и так называемых метасимволов. Метасимволы представляют собой символы, имеющие в регулярном выражении специальное значение:

```
\ . ^ $ | [ ( ) * + ? { }
```

Каждый метасимвол имеет свой смысл, например, он может использоваться для обозначения одиночного символа или группы символов, обозначать привязку к определенному месту строки, число возможных повторений отдельных элементов, возможность выбора из нескольких вариантов и т. д. Регулярное выражение строится с соблюдением определенных правил, причем в нем можно выделить операнды и операции.

Проанализируем назначение отдельных метасимволов. Так, открывающая и закрывающая круглые скобки `()` выполняют группирование элементов, а метасимвол `*` означает, что находящийся перед ним элемент или группа элементов могут встречаться один раз, много раз или не встречаться вообще. Метасимвол `+` указывает на то, что стоящий перед ним элемент или группа элементов встретятся хотя бы один раз, а знак вопроса `?` означает, что элемент может отсутствовать или встретиться только один раз.

Далее приводятся примеры регулярных выражений с использованием перечисленных метасимволов:

- `/fr.*nd/` — этому выражению удовлетворяют строки "frnd", "friend", "front and back";
- `/fr.+nd/` — этому выражению удовлетворяют строки "frond", "friend", "front and back", но не "frnd";
- `/10*1/` — этому выражению удовлетворяют строки "11", "101", "1001", "100000001";
- `/b(an)*a/` — этому выражению удовлетворяют строки "ba", "bana", "banana", "bananananana";
- `/flo?at/` — этому выражению удовлетворяют строки "flat" и "float", но не "flooat".

Метасимвол квадратные скобки (`[]`) указывает на группу отдельных символов. Примеры регулярных выражений с квадратными скобками:

- `[0123456789]` — этому выражению удовлетворяет любая цифра из диапазона 0—9;
- `[0-9]` — этому выражению удовлетворяет любая цифра из диапазона 0—9;

- `[0-9]+` — этому выражению удовлетворяет любая последовательность цифр;
- `[a-z]+` — этому выражению удовлетворяет любое слово из символов нижнего регистра;
- `[A-Z]+` — этому выражению удовлетворяет любое слово из символов верхнего регистра;
- `[ab n]*` — этому выражению удовлетворяет строка нулевой длины "", "`b`", любое количество пробелов, а также строка "`nab a banana`";
- `[^...]` — этому выражению удовлетворяют символы, не равные ". . .";
- `[^0-9]` — этому выражению удовлетворяет любой символ, не являющийся цифрой.

Диапазон значений, определяемых регулярным выражением, можно сузить, если использовать метасимволы фигурных скобок `{}`. Например, выражению `[0-9]{6}` удовлетворяет любая последовательность из 6-ти цифр, а выражению `[0-9]{6,10}` — любая последовательность от 6-ти до 10-ти цифр.

Метасимволы `^` и `$` означают начало и конец строки соответственно (символ `^` должен находиться за квадратными скобками `[]`). Вот как можно их комбинировать в регулярных выражениях:

- `/at/` — этому выражению удовлетворяют "at", "attention", "flat" и "flatter";
- `/^at/` — этому выражению удовлетворяют "at" и "attention", но не "flat";
- `/at$/` — этому выражению удовлетворяют "at" и "flat", но не "attention";
- `/^at$/` — этому выражению удовлетворяет только "at";
- `/^at$/i` — этому выражению удовлетворяют "at", "At", "aT" и "AT";
- `/^[ \t]*$/` — этому выражению удовлетворяет пустая строка или любая комбинация пробелов и символов табуляции.

Следует упомянуть об одной важной особенности использования символов `+`, `?`, `.`, `*`, `^`, `$`, `()`, `[]`, `{}`, `|`, `\`. Если перед ними стоит символ `\`, специальное назначение для этих символов ликвидируется, и они воспринимаются как литералы (в общем случае нежелательно также располагать символ `\` перед `/`). Следующие примеры демонстрируют эту особенность:

- `/10.2/` — этому выражению удовлетворяют "10Q2", "1052" и "10.2";
- `/10\.2/` — этому выражению удовлетворяют "10.2", но не "10Q2" или "1052";
- `/\*+/` — этому выражению удовлетворяют один или более символов `*`;
- `/A:\\DIR/` — этому выражению удовлетворяет "A:\DIR";
- `/\/usr\/bin/` — этому выражению удовлетворяет "/usr/bin".

Если символ \ предшествует алфавитно-цифровому символу, то такая комбинация имеет специальное назначение, представляя собой сокращенную форму выражения в квадратных скобках [].

Например, выражение \d эквивалентно [0-9], а \s обозначает пробел и эквивалентно выражению [\t\n\r\f]. Символ в таком выражении может быть представлен и шестнадцатеричным значением, например, \x1b означает esc-символ.

Легче всего манипуляции с регулярными выражениями понять на примерах.

Пример 1.

```
#!/usr/bin/perl -w
$str = "Quck brown   fox jumped    over lazy dog";
if ($str =~ /fox/)
 {
  print "fox is found\n";
 }
```

Здесь в строковой переменной $str ищется слово fox, и если оно обнаружено, то на экране печатается соответствующее сообщение. Для поиска (замены) выражения используется специальный оператор =~.

Пример 2.

```
#!/usr/bin/perl -w
$str = "Quck brown   fox jumped    over lazy dog";
$str =~ s/ +/+/gi;
print "$str\n";
```

В этом примере используется строковая переменная $str с тем же значением, что и в предыдущем примере. Цель программы — заменить все пробелы между словами данной строки на единичный символ +. Обратите внимание на оператор

```
$str =~ s/ +/+/gi;
```

Здесь символом s в правой части выражения обозначен оператор замены (substitution), а выражение " +" указывает на один или несколько следующих подряд пробелов.

Вот результат работы такой программы (она называется regexpr_demo2.pl):

```
# ./regexpr_demo2.pl
Quck+brown+fox+jumped+over+lazy+dog
```

Пример 3.

Это самый сложный пример по сравнению с предыдущими. Здесь на экран дисплея нужно вывести те элементы строкового массива `@str_array`, которые одновременно удовлетворяют двум условиям:

- строка не должна начинаться с цифры;
- строка не должна начинаться с литер "Fi".

Исходный текст примера представлен далее:

```
#!/usr/bin/perl -w
@str_array = ("First", "2-nd", "3-rd", "Fourth", "Fifth", "Sixth");

foreach (@str_array)
{
  print "$_\n" if (!/^\d/ && !/^Fi+/);
}
```

Результат работы программы (она называется regexpr_demo3.pl) будет таким:

```
# ./regexpr_demo3.pl
Fourth
Sixth
```

12.2.7. Обработка файлов и каталогов

Поскольку файлы занимают центральное место в обработке данных, то в языке Perl предусмотрен весьма широкий набор функций для манипуляций ими. При этом стандартные задачи (открытие файлов, чтение и запись данных) используют простые функции ввода/вывода, а для решения более сложных задач, таких, например, как асинхронная обработка данных и блокировка файла, разработаны и применяются более сложные функции. Здесь мы рассмотрим наиболее широко используемые функции для манипуляций файлами.

Доступ к файлам в Perl осуществляется посредством файловых манипуляторов — символических имен, представляющих данный файл в операциях чтения/записи. Эти символические имена на самом деле ссылаются на дескрипторы файлов, с которыми работает операционная система. Файловые дескрипторы используются во всех системных вызовах (`open()`, `read()`, `write()`, `close()`), но работать с ними сложнее, в то время как операции с файловыми манипуляторами не вызывают затруднений. Открыть файл можно с помощью функции `open`, имеющей синтаксис:

`open(`*`файловый_манипулятор`*`, "[`*`тип_операции`*`]`*`путевое_имя_файла`*`")`

Здесь *файловый_манипулятор* указывает на файловый манипулятор, *тип_операции* определяет, какая операция (чтение, запись, чтение/запись) будет выполняться над файлом, а *путевое_имя_файла* задает путь к файлу.

Если обозначить файловый манипулятор как `FH`, а путевое имя файла как `$path`, то функцию `open` можно представить в более доступной для понимания форме:

- `open(FH,"$path")` — файл открывается для чтения;
- `open(FH,">$path")` — файл открывается для записи;
- `open(FH,">+$path")` — файл открывается для чтения и записи;
- `open(FH,">>$path")` — файл открывается для дозаписи.

Содержимое файла можно прочитать одним из двух способов. В первом случае можно использовать следующий оператор:

```
@data = <FH>
```

Здесь `FH` — файловый манипулятор.

Во втором случае (который часто используется для чтения текстовых файлов с разделителями строк) можно читать строку за строкой в цикле `while`. Далее показаны оба способа чтения файлов.

Пример 1.

```
#!/usr/bin/perl -w
$file = shift or die "Usage: $0 path";
open(FH, "$file") or die "open: $!";
@data = <FH>;
close(FH);
print @data;
```

Программа выводит на экран содержимое файла, путевое имя которого задается в качестве единственного параметра командной строки. Этот параметр помещается в переменную `$file`. Если параметр командной строки отсутствует, программа завершается (функция `die`). Здесь `$0` — специальная переменная Perl, которая указывает на имя исполняемого файла программы.

По завершению файловых операций обязательно следует закрыть файловый манипулятор при помощи функции `close`, параметром которой является файловый манипулятор.

Пример 2.

```
#!/usr/bin/perl -w
$file = shift or die "Usage: $0 path";
```

```
open(FH, "$file") or die "open: $!";
while (<FH>)
{
  print ":$_";
}
close(FH);
```

В этом примере чтение содержимого файла и вывод на экран выполняются строка за строкой в цикле `while`, причем для этого используется переменная по умолчанию `$_`, в которую помещаются данные из файла.

Следующие два примера более сложные и показывают, как можно скопировать или переместить файлы.

Пример 3.

Программа копирования файлов представлена исходным текстом, показанным далее:

```
#!/usr/bin/perl -w
if ($#ARGV+1 != 2)
{
  die "Usage: $0 src dst\n";
}
$src = $ARGV[0];
$dst = $ARGV[1];
open(FSRC, "$src") or die "open: $!";
@data = <FSRC>;
close(FSRC);
open(FDST, "+>$dst") or die "open: $!";
print FDST @data;
close(FDST);
```

Программа принимает два параметра командной строки. Первый параметр соответствует зарезервированной переменной `$ARGV[0]` языка Perl и представляет собой путевое имя файла, который следует скопировать. Второй параметр является путевым именем файла, в который будет выполнено копирование (переменная `$ARGV[1]`). Файловый манипулятор файла-источника обозначен как `FSRC`, а файловый манипулятор файла-приемника — `FDST`. После чтения содержимого исходного файла в переменную `@data` выполняется запись данных в манипулятор `FDST`, для чего используется обычная функция `print`:

```
print FDST @data;
```

Смысл остальных операторов программы, думаю, понятен, и объяснять его не нужно.

Пример 4.

Программа перемещения файлов отличается от только что рассмотренной программы копирования лишь тем, что после завершения операции исходный файл удаляется при помощи функции `unlink`, принимающей в качестве параметра путевое имя файла-источника:

```
#!/usr/bin/perl -w
if ($#ARGV+1 != 2)
{
  die "Usage: $0 src dst\n";
}
$src = $ARGV[0];
$dst = $ARGV[1];
open(FSRC, "$src") or die "open: $!";
@data = <FSRC>;
close(FSRC);
open(FDST, "+>>$dst") or die "open: $!";
print FDST @data;
close(FDST);
unlink($src);
```

Мы рассмотрели основные методы работы с файлами в языке Perl. Хочу упомянуть еще одну функцию, позволяющую переименовать файл, — `rename`, имеющую синтаксис:

`rename(`*текущее_имя_файла*`, `*новое_имя_файла*`)`

Смысл параметров функции, думаю, понятен из их названия.

Следующие операторы позволяют выполнить проверки объектов файловой системы:

- `-e` *файл/каталог* — проверка существования файла/каталога;
- `-z` *файл* — проверка файла на предмет нулевого размера;
- `-d` *файл* — проверка, является ли объект файловой системы каталогом;
- `-s` *файл* — проверка размера файла (в байтах).

Рассмотрим особенности работы с каталогами в языке Perl. Напомню, что каталог представляет собой файл специального формата, помеченный в индексном узле как каталог. Каталог содержит информацию о других объектах файловой системы, находящихся в нем, и этим отличается от файла, содер-

жащего только двоичные данные. Для работы с содержимым каталога в языке Perl имеется несколько функций:

- `opendir(DIR, $имя_каталога)` — открывает каталог с именем *имя_каталога* и присваивает ему манипулятор каталога `DIR`;
- `readdir(DIR)` — читает содержимое каталога с манипулятором `DIR`;
- `closedir(DIR)` — закрывает каталог с манипулятором `DIR`.

Манипуляции с содержимым каталога рассмотрим на примере, исходный текст которого представлен далее. В этом примере на экран выводится содержимое рабочего каталога приложения, за исключением файлов "." и "..".

```
#!/usr/bin/perl -w
use Cwd;
$workdir = &cwd;
opendir(DIR, $workdir) or die "opendir: $!";
while (defined($file = readdir(DIR)))
{
  next if ($file =~ /^\./);
  print "$file\n";
}
closedir(DIR);
```

Чтение каталога и вывод информации осуществляются в цикле `while`, который выполняется до тех пор, пока не будет обработан последний элемент каталога. Оператор

```
next if ($file =~ /^\./);
```

пропускает вывод имени файла, если оно начинается с точки (.) или двоеточия (..).

По окончании обработки каталога он закрывается посредством функции `closedir`.

Заканчивая обзор файловых операций, хочу заметить, что ввиду ограниченности объема книги мы рассмотрели только часть возможностей языка Perl по работе с файлами и каталогами. Намного больше информации можно найти в документации, расположенной на различных интернет-сайтах, в частности, на **www.perl.org**.

12.2.8. Программные каналы

В языке Perl без особого труда можно создавать *программные каналы*, посредством которых можно получать вывод данных из других программ или направлять данные на вход приложений для дальнейшей обработки. Это кар-

динальным образом расширяет возможности Perl, позволяя создавать комплексные алгоритмы обработки данных, поручая часть работы другим приложениям. Программный канал можно организовать при помощи уже знакомой нам функции `open`. Например, для чтения вывода другой программы в программе на Perl можно использовать конструкцию вида

```
open(README, "программа аргумент1 аргумент2 ... |")
```

Здесь *программа* может представлять собой как команду операционной системы, так и пользовательское приложение. Вывод программы *программа* направляется в файловый манипулятор `README`. При использовании такой конструкции необходимо учитывать, что *программа* должна допускать перенаправление вывода.

Для передачи данных из приложения на Perl другой программе следует использовать конструкцию

```
open (WRITEME, "| программа аргумент1 аргумент2 ... ")
```

Здесь символ `|`, обозначающий программный канал, должен предшествовать имени программы *программа*, а сама *программа* должна допускать перенаправление ввода. Ввод данных в программу *программа* выполняется посредством файлового манипулятора `WRITEME`.

Рассмотрим несколько примеров работы с программными каналами.

Программа, исходный текст которой представлен далее, позволяет прочитать вывод команды `ls -l` и вывести данные на дисплей:

```
#!/usr/bin/perl -w
open(FD, "ls -l|") or die "open: $!";
@data = <FD>;
print @data;
close(FD);
```

Здесь функция `open` создает файловый манипулятор `FD` программного канала, после чего данные записываются в буфер, определяемый переменной `@data`. Затем содержимое буфера выводится на экран дисплея функцией `print`, а файловый манипулятор `FD` закрывается функцией `close`.

Если после оператора

```
@data = <FD>
```

поместить программный блок, выполняющий, например, сортировку или анализ содержимого файлов из полученного списка, то можно получить очень мощную программу-фильтр — и это только один из вариантов применения программного канала. Вообще, разумная комбинация команд UNIX

и программных конструкций языка Perl позволяет строить очень мощные по своей функциональности приложения.

Следующая программа демонстрирует передачу данных другому приложению:

```
#!/usr/bin/perl -w
open(FIND, "find . -name \"f*\" -print|") or die "open: $!";
@data = <FIND>;
close(FIND);
open(CPIO, "|cpio -pdu DIR") or die "open: $!";
print CPIO @data;
close(CPIO);
```

Здесь моделируется работа команды

```
find . -name "f*" -print | cpio -pdu DIR
```

которая копирует файлы текущего каталога, начинающиеся с литеры `f`, в каталог `DIR`. В программе используются две функции `open` — одна для приема данных, полученных командой поиска `find`, вторая — для передачи данных команде архивирования `cpio`. Как и в первом примере, данные временно сохраняются в буфере `@data`, и их можно обрабатывать по своему усмотрению, разработав соответствующий алгоритм.

Обратите внимание на оператор

```
open(FIND, "find . -name \"f*\" -print|") or die "open: $!";
```

Здесь в команде `find` присутствуют две наклонные черты `\` — это необходимо для подавления специальной функции символа `"`.

Обмен данными между приложениями можно выполнить и с помощью *именованных каналов*. Для этого следует создать при помощи команды `mkfifo` или `mknod` именованный канал:

```
# mknod /test.pipe p
```

В этом примере в корневом каталоге создается именованный канал `test.pipe` (пользователь должен иметь права root). После этого программы могут использовать `test.pipe` для обмена данными. Вот исходный текст программы, записывающей данные, вводимые с устройства стандартного ввода, в именованный канал `test.pipe` (в качестве устройства стандартного ввода в большинстве случаев выступает клавиатура):

```
#!/usr/bin/perl -w
open(PIPE, ">/test.pipe") or die "open: $!";
while (<>)
```

```
 {
  print "Enter string: ";
  if (/^(quit)$/)
     {
      print "Input Completed.\n";
      last;
     }
  print PIPE $_;
 }
close(PIPE);
```

Именованный канал открывается для записи функцией `open`. Далее приложение записывает введенные строки в файловый манипулятор `PIPE` до тех пор, пока не встретится строка `quit`, после чего канал закрывается для записи командой `close(PIPE)`(хотя при этом он может оставаться доступным для чтения).

Программа, выполняющая чтение данных из именованного канала `test.pipe`, представлена следующим исходным текстом:

```
#!/usr/bin/perl -w
open(PIPE, "</test.pipe") or die "open: $!";
while (<PIPE>)
  {
    print ">$_";
  }
close(PIPE);
```

Именованный канал `test.pipe` открывается для чтения функцией `open`, после чего данные можно читать из файлового манипулятора `PIPE`. По окончании чтения канал закрывается функцией `close(PIPE)`.

Последняя тема, которой мы уделим внимание, — сетевое программирование на языке Perl.

12.2.9. Сетевое программирование в Perl

Одной из замечательных особенностей языка Perl является возможность легкой разработки сетевых программ. Программный код таких приложений, в отличие, например, от языка C, намного читабельнее и скрывает многие детали программирования, что значительно облегчает программирование. Здесь будут вкратце рассмотрены общие аспекты сетевого программирования — более подробная информация находится в документации по языку Perl.

При разработке сетевой программы в нее следует включить модуль `Socket`, имеющий все необходимые функции (оператор `use Socket`).

Рассмотрим две программы. Одна из них является сервером TCP и работает с портом 5151, читая данные, передаваемые программой-клиентом, и выводя их на экран консоли. Обращаю внимание читателей на то, что как программа-сервер, так и программа-клиент работают на одной машине, используя интерфейс обратной связи с адресом 127.0.0.1, поэтому их легко проверить и отладить. Исходный текст программы-сервера представлен далее:

```
#!/usr/bin/perl -w
use Socket;
$host = inet_aton("127.0.0.1");
$port = 5151;
$addr = sockaddr_in($port, $host);
socket(SERVER, AF_INET, SOCK_STREAM, getprotobyname("tcp"))
      or die "socket: $!";
bind(SERVER, $addr) or die "bind: $!";
listen(SERVER, 10);
print "Waiting for requests...\n";
while (1)
{
  if (accept(NEW_CON, SERVER))
  {
    while (<NEW_CON>)
    {
      print "$_\n";
    }
    close(NEW_CON);
  }
}
close(SERVER);
```

В этой программе используются уже знакомые из предыдущих глав сетевые функции `socket`, `bind`, `accept` и `listen`. Эти функции имеют тот же смысл, что и соответствующие библиотечные функции языка C (в System V) или системные вызовы (BSD-совместимые системы). Хочу обратить внимание на то, что чтение данных из сокета выполняется так же, как и для обычных файловых манипуляторов.

Программа-клиент намного проще:

```
#!/usr/bin/perl -w
```

```
use Socket;
$host = inet_aton("127.0.0.1");
$port = 5151;
$addr = sockaddr_in($port, $host);
socket(CLIENT, AF_INET, SOCK_STREAM, getprotobyname("tcp"))
      or die "socket: $!";
connect(CLIENT, $addr) or die "connect: $!";
print CLIENT "Test String for TCP server\n";
close(CLIENT);
```

В программе-клиенте используются сетевые функции `socket` и `connect` с соответствующими параметрами. Передача данных программе-серверу выполняется посредством функции `print`.

Для проверки работы приложений следует открыть два окна консоли, после чего в одном из них запустить программу-сервер, затем в другом — программу-клиент.

Сокеты UNIX

Для обмена данными между процессами, работающими на локальной машине, эффективным средством является использование сокетов UNIX. Сокеты UNIX очень широко применяются в операционных системах UNIX при реализации клиент-серверных приложений, выполняющихся на одной машине, например, в системе X Window. Имена сокетов UNIX похожи на имена файлов, хотя реализованы они в виде специальных файлов. Принципы обмена данными с использованием сокетов UNIX такие же, как для обычных сетевых приложений — это означает, что прием/передачу информации можно осуществлять посредством сетевых протоколов TCP и UDP.

Рассмотрим пример простейшей реализации клиент-серверных приложений с использованием сокетов UNIX. Программа-сервер (ее можно назвать userver.pl) принимает данные на UNIX-сокете /mysock, функционируя в блокирующем режиме. Здесь, как и в сетевых приложениях, используются встроенные функции Perl для работы с сокетами, поэтому в исходный текст программы следует включить строку

```
use Socket;
```

Исходный текст программы-сервера приведен далее:

```
#!/usr/bin/perl -w
use Socket;
unlink("/mysock");
socket(SERVER, AF_UNIX, SOCK_STREAM, 0)
```

```
     or die "socket: $!";
bind(SERVER, sockaddr_un("/mysock")) or die "bind: $!";
listen(SERVER, 5) or die "listen: $!";
print "Local Server waits for requests ...\n";
while (1)
 {
   if (accept(NEW_CON, SERVER))
    {
     while (<NEW_CON>)
      {
       print $_;
      }
     close(NEW_CON);
    }
 }
close(SERVER);
```

Хочу обратить ваше внимание на несколько важных моментов. Поскольку многие приложения в процессе функционирования могут создавать специальные файлы, то не исключена вероятность (хоть и небольшая) того, что файл с именем /mysock уже существует. Вы должны удалить этот файл перед привязкой сокета функцией `bind`, для чего используется функция unlink. Если окажется, что такого файла нет, `unlink` никаких действий не предпримет, и программа продолжит выполнение.

В наличии специального файла /mysock можно убедиться, выполнив команду

```
# ls -l /mysock
srwxr-xr-x  1 root root 0 May 1 09:18 /mysock
```

При вызове функции `socket` вместо домена AF_INET следует указать AF_UNIX или PF_UNIX. Иногда в исходных текстах можно встретить обозначение AF_LOCAL, но не во всех системах это работает, поэтому лучше его избегать. Сокеты UNIX, как и сокеты Интернета, могут быть как дейтаграммными (SOCK_DGRAM), так и потоковыми (SOCK_STREAM). В данном случае программа-сервер работает с потоковыми сокетами. Еще одно отличие UNIX-сокетов от сетевых состоит в том, что для привязки сокета к специальному файлу используется адресная структура `sockaddr_un` вместо `sockaddr_in`.

Во всем остальном программа-сервер функционирует подобно сетевым программам, которые мы уже рассматривали. Характеристики соединения для

работающего процесса-сервера можно просмотреть при помощи утилиты `netstat`, выполнив командную строку (в Linux):

```
# netstat -a|grep Flags;netstat -a|grep mysock
Proto RefCnt Flags       Type       State         I-Node Path
unix  2      [ ACC ]     STREAM     LISTENING     83231  /mysock
```

Программа-клиент (она называется uclient.pl) передает строку сообщения на сервер, используя созданный программой-сервером специальный файл /mysock:

```
#!/usr/bin/perl -w
use Socket;
socket(CLIENT, AF_UNIX, SOCK_STREAM, 0)
     or die "socket: $!";
connect(CLIENT, sockaddr_un("/mysock"))
     or die "connect: $!";
print CLIENT "Test String for Local Server\n";
close(CLIENT);
```

Клиентская программа намного проще сервера — здесь не требуется прослушивать сокет и создавать новое подключение для приема вновь поступивших данных. Для установки соединения вызывается функция `connect`, после чего строка данных сразу же отправляется серверу посредством функции `print`.

Рассмотрим еще один пример использования сокетов UNIX, весьма полезный с практической точки зрения. Приложение-клиент передает текстовый файл программе-серверу, которая создает копию полученного файла с именем newfile на диске. Программа-клиент в качестве единственного параметра командной строки принимает имя передаваемого файла.

Исходный текст программы-сервера (ее можно назвать receive_file.pl) приведен далее:

```
#!/usr/bin/perl -w
use Socket;
unlink("/mysock");
socket(SERVER, AF_UNIX, SOCK_STREAM, 0)
     or die "socket: $!";
bind(SERVER, sockaddr_un("/mysock")) or die "bind: $!";
listen(SERVER, 5) or die "listen: $!";
print "Local Server waits for requests ...\n";
while (1)
 {
  if (accept(NEW_CON, SERVER))
```

```
   {
    open(NEW_FILE, ">newfile") or die "open: $!";
    while (<NEW_CON>)
     {
      print NEW_FILE $_;
     }
    close(NEW_FILE);
    close(NEW_CON);
    print "Received OK.\n";
   }
 }
close(SERVER);
```

Создание, привязку сокета и прием данных мы уже рассматривали в предыдущем примере, поэтому остановимся подробно на создании файла и записи в него данных. Для создания файла с именем newfile используется функция `open`, при успешном выполнении которой программе возвращается файловый манипулятор `NEW_FILE`:

```
open(NEW_FILE, ">newfile") or die "open: $!";
```

Запись полученных данных в файловый манипулятор выполняется в цикле `while (<NEW_CON>)` функцией `print`. По окончании передачи данных файловый манипулятор `NEW_FILE` закрывается функцией `close(NEW_FILE)`.

Программа-клиент представлена далее следующим исходным текстом:

```
#!/usr/bin/perl -w
use Socket;
$file = shift or die "Usage: $0 path\n";
socket(CLIENT, AF_UNIX, SOCK_STREAM, 0)
     or die "socket: $!";
connect(CLIENT, sockaddr_un("/mysock"))
     or die "connect: $!";
open(SRC, "<$file") or die "open: $!";
while (<SRC>)
 {
  print CLIENT $_;
 }
close(SRC);
close(CLIENT);
```

Здесь файл, подлежащий копированию, открывается для чтения с помощью функции

```
open(SRC, "<$file") or die "open: $!";
```

Далее содержимое файла передается серверу построчно в цикле `while (<SRC>)`, после чего файловый манипулятор следует закрыть. Хочу обратить ваше внимание на то, что программный код данного примера позволяет передавать и принимать содержимое только текстовых файлов, с разделителями строк в виде символов возврата каретки/перевода строки (CRLF). Для приема/передачи двоичных файлов требуется иная методика, которая здесь рассматриваться не будет, но она подробно описана в литературе по программированию на языке Perl.

12.2.10. Установка дополнительных модулей

Рассмотрим вкратце возможности обновления версий языка Perl. Кроме того, что Perl включается в дистрибутивы всех операционных систем, большое число разработчиков программного обеспечения выпускают отдельные пакеты программ для большинства операционных систем, в том числе и для UNIX. Более того, имеется возможность обновить установленную в операционной системе UNIX версию Perl, проинсталлировав дополнительные программные модули, предварительно загрузив их из Web-сайта **www.cpan.org**. На этом сайте собрана большая коллекция модулей, созданных независимыми разработчиками, и имеющая название CPAN.

Модули Perl распространяются в виде архивов tar.gzip, поэтому после загрузки их нужно предварительно распаковать программами gzip и tar. Предположим, что модуль загружен и распакован в каталог DIR. Тогда для установки модуля следует выполнить последовательность команд (с правами пользователя root):

```
cd DIR
perl Malefile.PL
make
make test
make install
```

Во время выполнения команд `make` обращайте внимание на диагностические сообщения — это поможет выявить проблемы на каждом этапе установки. Перед установкой модуля полезно также изучить документацию на него.

* * *

Заканчивая анализ возможностей разработки приложений в среде UNIX, хочу добавить, что самую свежую информацию по данной теме всегда можно обнаружить в Интернете.

Заключение

Система UNIX, будучи одной из самых первых разработок в области операционных систем, в настоящее время является самой мощной и динамичной операционной системой. Архитектура системы, разработанная еще в 70-е годы прошлого столетия, является оптимальной и в той или иной степени заимствована другими операционными системами, например, той же Windows.

С появлением Linux операционные системы UNIX приобрели массового пользователя, что обусловило резкое расширение сфер их применения. Массовое использование системы с открытым кодом, какой является Linux, обусловило лавинообразный рост разработок для этой операционной системы, начиная с середины 90-х годов XX века. В процессе своего развития ОС Linux приобрела массу дополнительных возможностей, включая возможность работы с графическим интерфейсом пользователя, обработки мультимедийных данных и т. д. Развитие этой операционной системы, в свою очередь, стимулировало развитие и других реализаций UNIX, как коммерческих, так и систем с открытым кодом. В этом контексте следует упомянуть о системе FreeBSD, ставшей довольно популярной и составляющей серьезную конкуренцию Linux.

В настоящее время сфера применения операционных систем UNIX постоянно расширяется. Кроме областей, где операционные системы UNIX используются чаще остальных (интернет-серверы, коммуникационные системы, военные и специальные разработки), появились и новые сферы применения. Благодаря развитию офисных приложений для UNIX, таких, например, как StartOffice, пакетов для обработки графики и видео и т. д., рабочие станции под управлением этой операционной системы стали понемногу вытеснять Windows-совместимые системы с установленным программным обеспечением Microsoft Office.

Что же касается перспектив развития операционных систем UNIX, то сделать какие-либо прогнозы на отдаленное будущее трудно. Тем не менее, можно

смело предположить, что архитектура UNIX-систем, в отличие от Windows, далеко не исчерпала своих возможностей и имеет значительный ресурс для развития. Также очевидно, что основной проблемой быстро растущего рынка операционных систем UNIX является стандартизация. В краткосрочной перспективе операционные системы UNIX будут развиваться в сторону поддержки новых технологий обмена данными, оставаясь наиболее популярной платформой для интернет-серверов и систем телекоммуникаций. Не следует исключать и постепенное расширение парка офисных систем на базе свободно распространяемых операционных систем Linux и FreeBSD, учитывая тенденции в области развития пакетов программ для обработки текстовых и графических документов.

Возможность создания дистрибутивов операционных систем UNIX, занимающих небольшой объем дискового пространства, делает перспективным применение UNIX в мобильных устройствах и в сфере автоматизации производственных процессов, учитывая значительный прогресс в области твердотельной электроники, позволяющей создавать запоминающие устройства небольших размеров и большой емкости.

Предметный указатель

(окончание рубрики см. на стр. 516)